Étude et développement d'une bibliothèque d'adaptation du parallélisme neuromimétique au parallélisme MIMD

Yann Boniface

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAE + M

Etude et développement d'une bibliothèque d'adaptation du parallélisme neuromimétique au parallélisme MIMD

THÈSE

présentée et soutenue publiquement le 23 octobre 2000

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Yann Boniface

Composition du jury

Président : Dominique Méry, Professeur, Université Henri Poincaré.

Rapporteurs : Hélène Paugam-Moisy, Professeur, Université Lumière Lyon 2.
Jeanny Hérault, Professeur, Université Joseph Fourier, Grenoble.

Examinateur : Olivier Coulaud, Directeur de Recherche, INRIA.

Directeu Directeur de Recherche, INRIA.
ef de travaux, Supélec.

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Résumé

Les réseaux de neurones artificiels sont connus pour être des modèles intrinsèquement parallèles. De plus ces modèles réclament une puissance de calcul toujours plus conséquente, notamment pour les modèles d'inspiration biologique du domaine. Dans le même temps, la technologie des ordinateurs parallèles devient de plus en plus accessible tant du point de vue de leur disponibilité que du point de vue financier.
Il apparaît donc intéressant d'utiliser les ordinateurs parallèles pour résoudre les problèmes de coûts d'exécution et de taille des réseaux de neurones posés aux connexionnistes.

Cette thèse présente un nouveau simulateur parallèle de réseaux de neurones artificiels. Notre simulateur se veut tout à la fois un outil d'aide au développement des réseaux de neurones artificiels et un outil permettant d'accélérer leurs exécutions. Il doit ainsi faciliter les implantations des réseaux, l'utilisation de réseaux de grandes tailles et l'étude du parallélisme des réseaux de neurones et des phénomènes d'émergences issus de grandes populations de neurones biologiques.

Avant de développer ce simulateur nous effectuons une étude comparée des parallélismes des réseaux de neurones artificiels et des ordinateurs parallèles modernes les plus courants. Nous mettons ainsi en valeur les grandes différences entre ces deux modèles de parallélisme puis nous présentons nos solutions pour offrir aux connexionnistes un outil d'aide au développement de réseaux de neurones utilisant les propriétés parallèles des modèles connexionnistes pour permettre leurs exécutions sur machines parallèles sans modifications des algorithmes dans ce sens.

Notre simulateur se présente donc sous la forme d'une bibliothèque de fonctions sur le langage 'C', permettant l'étude du parallélisme neuronal et les exécutions sur machines séquentielles classiques comme parallèles de type MIMD à mémoire partagée.

Mots-clés: Réseaux de neurones, Connexionnisme, Parallélisme, mémoire partagée, Simulateur, Bibliothèque de fonctions, MIMD

Abstract

Artificial neural networks are known to be intrinsic parallel models. Moreover, they need large computation capabilities, especially for biologically inspired models. In the same time, parallel computer technologies get more accessible, in terms of availability as in terms of financial costs.

So it seems interesting to use parallel computers to solve the neural networks difficulties : computation costs and sizes of the networks.

This work presents a new parallel simulator dedicated to artificial neural networks. This simulator aiming at facilitatting the development of neural networks models. It also allows to study the parallel properties and the emergence capabilities of large population of artificial neurons.

To build this simulator, we study and compare both neural networks parallelism and general purpose computer parallelism. We show large differences between these parallelisms and propose a solution to map the neural parallelism in the computer parallelism. This mapping allows to propose a simple tool for connectionists. This tool allows connectionists to use MIMD shared memory parallel computer without any parallel specific knowledge, to implement their model without any algorithmic modifications.

In its form, the simulator is a function library in 'C' language. This simulator works in both sequential workstation computers and MIMD shared memory general purpose parallel computers.

Remerciements

Comme pour tout ce qui touche à la recherche, le travail présenté ici n'est que la partie la plus austère de multiples collaborations. Je souhaite profiter de cette espace pour saluer les différents protagonistes qui ont contribué, chacun à leur manière, à l'aboutissement de ce travail.

Une première pensée pour les membres du jury qui, par l'intérêt porté aux travaux et la qualité de leurs remarques, ont su rendre enrichissante et somme toute agréable (ce qui n'était nullement gagné....) l'étape la plus protocolaire de ces années.

De sa genèse à sa conclusion, ce travail est aussi celui de mes encadrants : Stéphane et Frédéric qui m'ont constamment soutenu, encouragé et guidé par leurs compétences et l'intérêt qu'ils ont su porter à ce sujet. Je n'oublie pas Nicolas, premier interlocuteur lorrain, qui m'a proposé ce sujet et fait venir, me permettant ainsi de sortir d'une impasse.

La partie la plus marquante de ces années de doctorat sera sans contexte l'immersion dans l'équipe Cortex et les relations avec ses membres. Je pense plus particulièrement à Hervé, Nicolas et Laurent, qui ont pris une part prépondérante dans chacun des aspects de mon travail. La force de ces animateurs de road-movie est avant tout d'avoir, entre autres apports à la science, institutionnalisé le n'*importe quoi* et d'en avoir fait une dynamique, sous le regard attéré du mandarin. Il leur a fallut toutefois, pour parvenir à leurs fins, totalement sacrifier la notion d'humour.

Je n'oublie pas les autres membres de cette équipe, des anciens partis prêcher dans d'autres paroisses, aux actuels et nouveaux auquels je souhaite une continuité et que je remercie pour leur aide précieuse durant les longs mois de rédaction et les périodes de répétition. Dans ce cadre, il me faut adresser un grand merci à tout l'équipe de relecteurs qui a finallement réussi à reconcilier ce manuscrit avec la langue française.

Ces remerciements seraient inutiles s'ils ne permettaient de rendre hommage à tous ceux qui ont représenté mon quotidien tout au long de ces années et avec lesquels nous avons longuement démontrer que si culture et révolution sont de vrai concepts, leur conséquence la plus visible se situe au niveau du pouvoir d'achat des marchands de vins et spiritueux. En plus de la bise à Poulette, je salue donc tous les membres actifs de ces interminables débats, avec une pensée plus precise pour les membres du HIBS, qui ont su, douloureusement, épurer la notion de soirée conviviale en s'affranchissant des concepts pour ne plus se consacrer qu'a l'essentiel. Je n'oublie pas tous les autres, ceux d'avant les années nancéennes, qui avaient déjà beaucoup fait pour me forger cette sensibilité et cette délicatesse qui a permis mon intégration dans les milieux sus-cités.

Table des matières

Table des figures

Introduction

Nous présentons dans ce manuscrit un nouvel outil d'implantation de réseaux de neurones artificiels. Cet outil souhaite tout à la fois faciliter les phases de développement et de validation de ces modèles neuronaux et leur offrir la puissance de calcul des machines parallèles modernes. Nous proposons au monde connexionniste l'utilisation des caractéristiques *intrinsèquement parallèles* de leurs modèles à la fois pour les développements et les implantations de leurs modèles, mais aussi dans un but d'étude de ces caractéristiques. L'étude des caractères émergents d'un réseau vu comme une somme de neurones concurrents peut, par exemple, permettre d'envisager de nouvelles architectures. La possibilité de disposer de machines parallèles doit permettre non seulement d'accélérer les applications actuelles, mais aussi d'envisager de nouvelles architectures connexionnistes, plus complexes, de plus grande taille ou travaillant sur de plus grosses bases de données.

Issu de l'*Intelligence Artificielle*, science souhaitant modéliser l'intelligence à l'aide de l'outil informatique, le connexionnisme[1] puise ses fondements dans la plupart des sciences du vivant telles que la neurobiologie, la biochimie, la psychologie, la physiologie, tout en s'appuyant sur de solides formalismes mathématiques.

Cette science souhaitait s'inspirer des observations et des études sur les mécanismes de fonctionnement du cerveau pour construire de nouveaux paradigmes calculatoires. Si les ordinateurs furent, à l'origine, conçus comme des cerveaux artificiels par Von Neumann [von Neumann, 1958], les capacités des deux modèles s'avèrent actuellement bien différentes. Les ordinateurs séquentiels, dits à architecture de *von Neumann*, se montrent très performants pour résoudre de gros calculs ou des problèmes très bien définis, problèmes pouvant se simuler sous forme de calculs, mais pour des problèmes moins bien définis, ils s'avèrent bien inefficaces, tandis que l'homme -et même l'animal- les résolvent de manière évidente.

De plus en plus présent et reconnu dans l'informatique, le connexionnisme apporte maintenant ses solutions dans des domaines extrêmement variés. Tout d'abord, indépendamment de son inspiration originelle des neurosciences, le connexionnisme a principalement évolué aujourd'hui vers des modèles de traitement de données proches des statistiques. Les tâches effectuées sont alors relatives à la reconnaissance des formes, au diagnostic,à la prédiction, au contrôle et même, plus récemment, à la fouille de données : les domaines d'application touchent tout le monde socio-économique qui nous environne. Ensuite, le connexionnisme poursuit l'exploitation de l'inspiration biologique avec la mise au point de modèles dont les structures et les fonctions correspondent plus précisément au monde du vivant. Les tâches sont alors relatives à la perception et à la cognition animale et humaine et les domaines d'application sont aussi bien technologiques

1. Le terme *connexionnisme* sera utilisé dans ce manuscrit comme terme générique représentant l'étude des *réseaux de neurones artificiels.*

(robotique) que relatifs à ces domaines d'inspiration (modèles utilisables par des biologistes, psychologues, médecins, etc.). Dans tous les cas, les modèles connexionnistes sont particulièrement appréciés pour leurs capacités de *généralisation* et de *résistance au bruit*, propriétés héritées de leur paramétrage par *apprentissage*. Il est indéniable qu'ils ont aujourd'hui acquis une place reconnue comme outils numériques, adaptatifs et distribués de traitement de l'information, principalement dans les domaines technologiques relatifs aux sciences de l'ingénieur, mais aussi de plus en plus comme outils d'étude et de simulation dans le monde des sciences du vivant.

Si les champs d'application et de recherche de cet axe scientifique sont en constante expansion, les réseaux restent difficiles à utiliser, principalement en raison des temps de calcul attachés à ces modèles. Ces problèmes de temps de calcul sont particulièrement saillants lors de l'apprentissage des paramètres des réseaux. Un réseau contenant de nombreux paramètres nécessite un espace d'exemples de grande taille pour son apprentissage et cet apprentissage peut ainsi durer jusqu'à plusieurs jours. De plus, le manque de méthodes formelles pour la détermination d'architectures adaptées entraîne un temps non négligeable de mise au point et de validation des topologies de réseaux. Pour finir, les réseaux implantés sont toujours plus grands, ils nécessitent un nombre de plus en plus important de paramètres pour s'adapter à des problèmes de plus grande ampleur. Dans le cas des modèles biologiques, ils simulent des propriétés de plus en plus complexes, simulation qui passe par des nombres élevés d'unités au sein des réseaux, unités qui, elles-mêmes, se complexifient.

Ce problème de temps d'exécution est une des principales limitations à l'utilisation des modèles connexionnistes.

L'utilisation du parallélisme informatique fut rapidement envisagée pour pallier ce problème et réduire les temps de calcul. Il semblait notamment intéressant de paralléliser la phase d'apprentissage des réseaux et d'utiliser ensuite les réseaux paramétrés sur des supports plus courants.

En plus de l'habituelle recherche d'accélération des applications inhérente à l'utilisation de l'informatique parallèle, l'utilisation du parallélisme semblait évidente au regard des topologies connexionnistes. A l'image de son modèle biologique, le cerveau, le connexionnisme s'appuie sur les capacités de calcul distribué d'une somme d'unités élémentaires fortement connectées. Par ces propriétés, les réseaux de neurones artificiels sont souvent considérés comme des modèles de calcul parallèle (par exemple un des ouvrages de référence du domaine est le *Parallel Distributed Processing* de Rumelhart et son équipe [Rumelhart *et al.*, 1986]). Le parallélisme prêté aux réseaux connexionnistes devait donc pouvoir être aisément plaqué sur le parallélisme des super-ordinateurs. On ne compte plus les publications du domaine qui, notant le *parallélisme intrinsèque* des réseaux de neurones artificiels, citaient comme perspectives aux travaux présentés une "évidente" parallélisation du modèle, qui accélérerait les calculs ou permettrait d'implanter des modèles de plus grande taille.

De la théorie à la pratique, il restait un pas.... que nous sommes toujours en train d'essayer de franchir.

En effet, l'expérience a montré que la nature parallèle des réseaux connexionnistes se mariait assez mal aux contraintes du parallélisme informatique, logiciel comme matériel. Les difficultés rencontrées sont dues aux caractéristiques des réseaux de neurones en matière de *communication* entre les unités. Ces communications sont en effet brèves, peu d'informations sont échangées au cours d'une communication, mais extrêmement fréquentes, le tout sur des réseaux massivement connectés. Ces spécificités des réseaux de neurones s'avèrent très éloignées du parallélisme actuel des ordinateurs, plutôt basé sur de longues périodes de calculs concurrents séparées par de

brèves, mais efficaces, périodes de communication entre processeurs.

Ces difficultés, ajoutées à la mouvance technologique du parc des machines parallèles, ont fortement limité les modèles de parallélisation des réseaux de neurones artificiels. Il existe ainsi de nombreuses architectures connexionnistes parallélisées sur une machine parallèle, mais l'utilisation courante, de la part de la communauté connexionniste, des machines parallèles pour implanter, tester et modifier ses modèles ne s'est pas faite. La règle est de confier, à des fins d'accélération, la parallélisation d'architectures abouties à des spécialistes du parallélisme, ces spécialisations étant avantageusement effectuées sur architectures matérielles dédiées (cartes VLSI, FPGA, etc.).

Les augmentations en taille des réseaux, surtout les réseaux d'inspiration biologique, montrent pourtant les intérêts que pourrait apporter un outil d'utilisation aisée de la puissance des machines parallèles *par les connexionnistes eux-mêmes*. En effet, dans les simulations de réseaux d'inspirations biologiques, le grand nombre d'unités du modèle et leurs interrelations multiples doivent être totalement simulées pour espérer voir un jour le résultat d'une exécution.

L'utilisation du *parallélisme neuronal* et des phénomènes d'émergence qui lui sont attachés semblent être actuellement une des limites au développement de modèles complexes d'inspiration biologique, limite imposée par la "faible" puissance de calcul offerte par les ordinateurs classiques en comparaison de la puissance de la moindre aire corticale.

Notre travail intervient donc à ce stade : nous proposons un outil, une bibliothèque de développement de réseaux de neurones, permettant aux connexionnistes d'étudier et d'implanter aisément leurs modèles, tout en offrant l'opportunité d'exécuter efficacement les réseaux sur machines parallèles courantes. Pour favoriser les implantations, notre simulateur utilise un formalisme proche du formalisme biologique, incluant les aspects d'exécution concurrentes des unités des modèles. L'utilisateur peut donc implanter des réseaux de grande taille et étudier les propriétés de parallélisme des réseaux de neurones artificiels.

Pour obtenir ce résultat, nous avons mené une étude conjointe des deux parallélismes concernés, parallélisme informatique et parallélisme connexionniste. Cette étude nous a permis de modéliser le formalisme d'implantation neuronal proposé et les techniques de programmation parallèle utilisées. Le formalisme neuronal doit être à la fois utile, il doit faciliter les implantations de réseaux, et convivial, il doit pouvoir utiliser les algorithmes classiques du connexionnisme. L'implantation parallèle doit permettre d'obtenir de bonnes performances sur tous types de réseaux de neurones artificiels, et plus particulièrement sur les réseaux biologiques. Pour finir, le formalisme de programmation proposé aux utilisateurs doit parfaitement masquer l'utilisation du parallélisme matériel, et permettre ainsi l'utilisation de machines parallèles sans connaissances spécifiques dans ce domaine.

Nous expliciterons dans le manuscrit les caractéristiques de notre simulateur et les choix qui ont motivé ces caractéristiques.

Notre bibliothèque étant destinée à simuler des réseaux connexionnistes, nous présenterons tout d'abord les différents aspects topologiques et algorithmiques de ces modèles. Le chapitre 1 présente les aspects attachés aux unités de base des réseaux. Le chapitre 2 présente les caractéristiques de réunion de ces unités de base en réseaux.

Notre simulateur devant permettre les exécutions sur machines parallèles, nous présenterons ensuite les aspects liés à l'utilisation de cette technologie informatique.

Le chapitre 3 présente les contraintes et spécificités liées à la programmation sur machines

parallèles. Le chapitre 4 détaillera les aspects plus technologiques des différentes architectures matérielles et les contraintes supplémentaires apportées par celles-ci à la programmation. Cette partie devra justifier les choix concernant les aspects parallèles de notre simulateur, choix de la machine hôte des exécutions parallèles des réseaux implantés et choix techniques de développement de la bibliothèque.

Le chapitre 5 présente l'existant en matière de parallélisation et de simulation de réseaux connexionnistes. Nous détaillons, dans ce chapitre, le *parallélisme intrinsèque* des réseaux de neurones, nous présentons les implantations parallèles spécifiques existantes et, pour introduire les outils de parallélisation *génériques* de réseaux connexionnistes, nous nous attardons sur les différents types de simulateurs existants. Ce chapitre nous permettra de nous situer dans les simulateurs de réseaux de neurones artificiels, de justifier le type de parallélisme neuronal utilisé et les choix d'implantation matérielle de la bibliothèque.

La deuxième partie présente plus précisément notre contribution, nous y présentons notre bibliothèque, ses justifications, ses apports et son utilisation.

Le chapitre 6 présente et argumente les formalismes choisis en terme de connexionnisme comme en terme de parallélisme. L'argumentation de ces choix s'appuie sur les constatations effectuées au cours de la première partie du manuscrit. Nous présentons aussi les différences entre ces deux formalismes, différences que notre travail consistera à masquer.

Le chapitre 7 est consacré à une présentation du simulateur en tant qu'outil. Nous présentons la méthode de programmation de réseaux à utiliser et les fonctions offertes. Ce chapitre est illustré par un exemple, l'implantation de l'algorithme du *jeu de la vie.*

Le chapitre 8 précise l'implantation sur machine parallèle de notre bibliothèque et les différentes étapes de l'exécution d'un réseau. Nous présentons dans cette partie les solutions choisies en matière d'implantation parallèle. Ce chapitre reprend les points évoqués dans le chapitre 3 et présente notre position sur ces points.

Pour finir, nous présentons dans le dernier chapitre, le chapitre 9, un exemple d'implantation détaillé de réseau, une carte de Kohonen, et quelques performances d'autres architectures représentatives. Ces performances sont les temps d'exécutions de trois architectures développées à l'aide de notre simulateur, et exécutées sur machine parallèle.

Première partie

Les fondements scientifiques

Chapitre 1

Du neurone biologique au neurone artificiel

Nous nous intéresserons, dans cette partie du manuscrit, aux aspects fondamentaux des réseaux de neurones, en nous inspirant essentiellement des présentations faites dans les ouvrages de Hervé Abdi [Abdi, 1994], de Claude Touzet [Touzet, 1992] et de John Hertz [Hertz *et al.*, 1991]. Néanmoins, nous ne chercherons pas à effectuer un inventaire exhaustif des multiples facettes de ce champ d'étude, ni à détailler les différents postulats et théorèmes sur lesquels il s'appuie. Cette présentation a pour principal objectif de cerner le domaine afin d'introduire et de justifier les choix ayant guidé nos travaux en matière d'outils de simulation. Nous reprendrons donc des choses connues et maintes fois présentées du domaine, mais qui sont incontournables pour modéliser un simulateur. Nous tenterons donc un parcours représentatif du domaine des réseaux de neurones. Ce parcours nous permettra tout à la fois de décrire les caractéristiques principales des réseaux de neurones et de déterminer les outils nécessaires au développement de ces réseaux. Ces informations nous serviront ensuite à modeler notre outil de simulation tant dans sa forme algorithmique que dans son interface à l'utilisateur. Pour des descriptions plus précises du domaine des réseaux de neurones et de son histoire, le lecteur pourra se reporter aux livres de Cowan [Cowan, 1989], de Minsky et Papert [Minski et Papert, 1969], de Arbib [Arbib, 1995] ou de Rumelhart et McClelland [Rumelhart *et al.*, 1986], comme à l'article de Medler [Medler, 1998].

Nous nous intéresserons tout d'abord aux différentes formes des unités de base du domaine, les *neurones*, et aux algorithmes qui les régissent. Nous verrons ensuite les formes d'assemblage les plus courantes de ces unités permettant de construire explicitement les réseaux connexionnistes. Nous présenterons plus particulièrement les types de neurone utilisés et les topologies créées par les connectivités choisies. Nous évoquerons aussi l'apprentissage au sein des neurones et des réseaux de neurones artificiels. Bien que cet aspect du connexionnisme n'influence que très peu notre travail, il est l'une des grandes caractéristiques du domaine et, à ce titre, influence grandement les choix topologiques et algorithmiques.

1.1 Le modèle biologique

Vouloir s'inspirer du fonctionnement du cerveau entraîne naturellement à s'inspirer de son élément de base : *le neurone biologique*. Le cerveau peut en effet être vu comme la somme de plusieurs milliards de neurones fortement connectés.

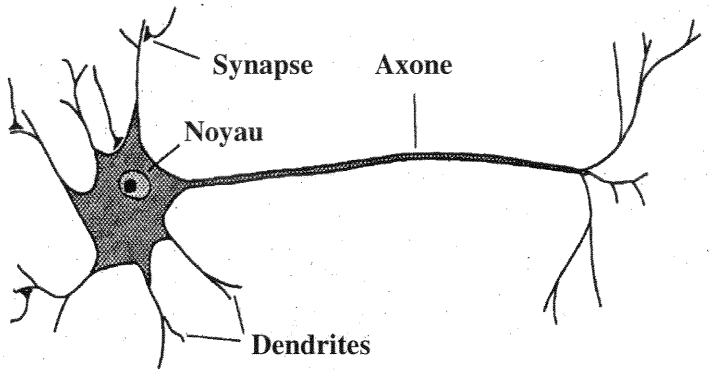


FIG. 1.1 – *Une représentation simplifiée d'un neurone biologique. D'après [Hertz* et al., *1991]*

Un neurone biologique est une unité relativement simple disposant de capacités extrêmement limitées. C'est leur grand nombre, associé à une multitude de connexions qui crée la complexité et la puissance du cerveau.

Une description succincte du neurone biologique le représenterait comme une entité recevant de l'information, des activations, de ses congénères, par l'intermédiaire de connexions, les *synapses*. Le neurone traite ces informations au niveau du noyau pour définir son activation. Celle-ci est à son tour transmise, lorsque le neurone *décharge*, aux neurones qui lui sont connectés par l'intermédiaire de son *axone* (voir figure 1.1).

Au niveau topologique, un neurone est *connecté* à un autre neurone quand l'une de ses dendrites est en relation, par l'intermédiaire d'une *synapse*, avec l'axone de ce dernier.

Chaque neurone biologique possède un **unique axone**, et **plusieurs synapses**. Le nombre de synapses peut être de plusieurs milliers pour un même neurone. La transmission de l'information est effectuée à travers les axones, des synapses et des dendrites. Elle se fait par flux électrochimiques créant des potentiels d'actions appelés les *décharges* du neurone [Stevens, 1978].

Ces neurones biologiques sont à la base de tout le système cérébral, les sous-divisions du cerveau, comme les colonnes ou les aires corticales, n'étant que des composés de neurones répondant à une topologie précise.

Suivant ce modèle, le connexionnisme cherche à développer de nouveaux paradigmes computationnels en construisant des réseaux de neurones artificiels, topologies développées à partir de la modélisation d'une simplification du neurone biologique.

1.2 Les neurones artificiels

Construire des réseaux de neurones artificiels consiste, pour l'essentiel, à définir un certain nombre d'unités de base, puis à construire une topologie de réseau en créant des communications entre ces unités.

Il existe cependant différents types d'unités de base, certaines créées pour approcher au mieux les caractéristiques du modèle biologique, d'autres, de manière plus pragmatique, développées afin d'étendre les capacités de l'unité artificielle.

1.2.1 Le neurone formel de Mc Culloch et Pitts

C'est d'une simplification du neurone biologique équivalente à celle que nous avons présentée dans la partie 1.1 que McCulloch et Pitts se sont inspirés pour créer, en 1943, leur *neurone formel* [McCulloch et Pitts, 1943], qui est le premier neurone artificiel de l'histoire. Ce modèle théorique, qui est à l'origine du neuromimétisme, définit une modélisation mathématique du neurone biologique, un modèle numérique visant à représenter le fonctionnement neuronal.

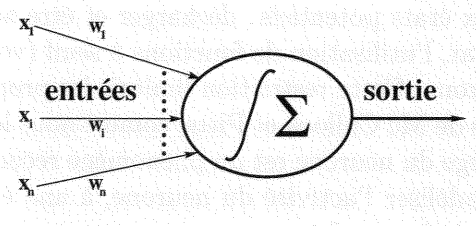


FIG. 1.2 – *Un neurone formel*

Le neurone formel, tel que représenté dans la figure 1.2, se veut donc une modélisation très simplifiée du modèle biologique. A l'image de ce grand frère naturel, il peut posséder plusieurs entrées, simulant les dendrites, pour le relier aux autres neurones du système. Il reçoit, par ses entrées, des flux excitateurs ou inhibiteurs, modélisés numériquement sous la forme d'une valeur positive ou négative. Il détermine, périodiquement, son activation à partir de ces entrées. C'est cette activation, en fonction de sa valeur numérique, qui déterminera l'état de la sortie du neurone, sortie qui représente le rôle de l'axone du neurone biologique.

Détermination de la sortie du neurone Dans ce premier modèle de neurone artificiel, la valeur de sortie du neurone, qui correspond grossièrement à la notion de *décharge* du modèle biologique, est binaire. C'est en fonction de sa valeur d'activation, donc de ses entrées et de ses poids, que le neurone déchargera ou non.

En terme de modélisation, un *poids* est attribué à chacune des entrées du neurone, ce qui permet de hiérarchiser l'importance des connexions.

Pour déterminer son activation, le neurone formel effectue la somme pondérée de ses entrées, puis il détermine la valeur de sa sortie, binaire, par l'intermédiaire d'une *fonction à seuil*, comme décrit par l'équation 1.2. Si cette activation dépasse le seuil, le neurone décharge, et la valeur de sa sortie est 1. Dans le cas contraire il est considéré comme inactif et sa sortie est nulle. C'est cette valeur qui sera transmise aux neurones qui lui sont connectés. Il est donc clair que, dans ce modèle, les entrées aussi sont binaires.

Pour un neurone formel i, ayant n entrées, chaque entrée x_j étant associée à un poids w_{ij}, et utilisant la fonction de transfert f pour déterminer son activation, le calcul de la sortie S_i se détermine par l'intermédiaire des équations :

$$S_i = f(\sum_{j=1}^{n} w_{ij}.x_j) \tag{1.1}$$

avec

$$f(x) = \begin{cases} 0 & \text{si } x \leq seuil \\ 1 & \text{si } x > seuil \end{cases} \tag{1.2}$$

1.2.2 Le neurone continu

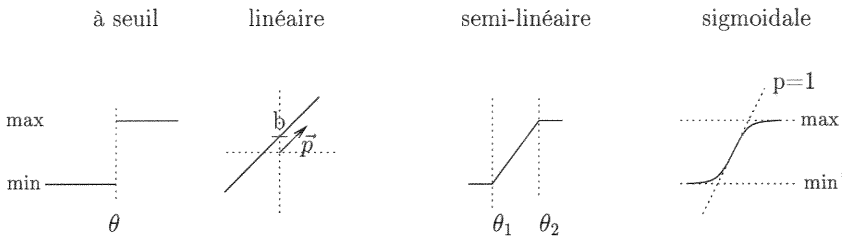
Dans le modèle d'origine du neurone formel, les activités des neurones n'étaient que binaires. Le neurone n'a donc que deux états potentiels : décharger et être actif, ne pas décharger et être inactif. Pour obtenir ce résultat, l'utilisation de fonctions à seuil (voir figure 1.3) suffisait comme fonctions de transfert du neurone. Cette restriction limitait les propriétés et les applications des réseaux basés sur les neurones de Mc Culloch et Pitts, comme nous le verrons dans la partie 2.2.5. De plus, si la notion de décharge du neurone est un phénomène reconnu des neurones biologiques, il peut être intéressant de modéliser l'activité du neurone, à une échelle de temps plus grande, par sa fréquence moyenne de décharge, fréquence moyenne qui est une valeur continue.

Pour abolir cette restriction, les neurones de Mc Culloch et Pitts ont donc été modifiés afin de permettre une sortie continue, créant ainsi la notion de *neurone continu.*

Pour obtenir une sortie continue tout en gardant la philosophie inhérente au modèle d'origine, la fonction générale de calcul de l'activité est devenue :

$$S_i = f\left(\sum_{j=1}^{n} w_{ij}.x_j - \beta\right) \tag{1.3}$$

Dans cette fonction d'activation, le seuil de déclenchement est intégré sous forme d'un biais β, la fonction de transfert n'est maintenant plus uniquement une fonction à seuil, elle varie selon les problèmes à résoudre. Les fonctions les plus fréquemment usitées, avec ce neurone continu, ne sont plus les fonctions à seuil mais les fonctions semi-linéaires, linéaires ou sigmoïdales, comme présentées dans la figure 1.3, mais toute fonction continue et croissante peut-être utilisée.



$$g(x) = \begin{cases} min & \text{si } x < \theta \\ max & \text{si } x > \theta \end{cases} \quad g(x) = px + b \quad g(x) = \begin{cases} min & \text{si } x \leq \theta_1 \\ px + b & \text{si } \theta_1 < x < \theta_2 \\ max & \text{si } \theta_2 \leq x \end{cases} \quad g(x) = \frac{max - min}{1 + \exp(-px)} + min$$

FIG. 1.3 – *Les quatre fonctions de transfert les plus fréquemment utilisées dans les neurones artificiels*

1.2.3 Le neurone impulsionnel

L'objectif est ici de s'approcher du modèle biologique en matière de traitement de l'information. Au niveau biologique l'information est transmise sous forme de modulation de fréquence. L'information, l'activité, du neurone est contenue dans la suite de potentiels d'actions émis par ce neurone [Gerstner, 1998] :

$$F_i = \{t_i^{(1)},...,t_i^{(n)}\} \tag{1.4}$$

De nombreuses modélisations de ce codage existent [Maass et Bishop, 1998]. Elles prennent en compte, par exemple, la fréquence des décharges des neurones ou l'ordre de réception de celles-ci et les utilisent pour reconstruire l'information ou synchroniser une population de neurones.

Ce formalisme présente l'avantage de s'approcher au plus près des paradigmes biologiques, construits sur de l'information reçue en temps continu. La modélisation informatique de ce temps continu nécessite une discrétisation à faible pas de temps, et une synchronisation des différentes unités sur ce pas de temps. Ces contraintes rendent les applications chères en temps de calcul. Elle sont donc actuellement limitées à des réseaux contenant un très faible nombre d'unités.

1.2.4 Le neurone à fuite

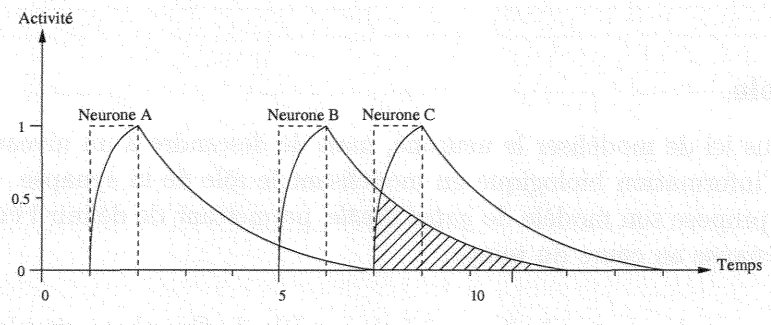


FIG. 1.4 – *Neurones à fuite. L'activité des neurones est prolongée dans le temps, ce qui permet de corréler des évènements impossibles à détecter avec des décharges classiques (ici en pointillés). D'après [Frezza-Buet* et al.*, 2000]*

Pour donner un caractère temporel aux neurones artificiels, Reiss et Taylor [Reiss et Taylor, 1991] ont proposé les *neurones à fuite*, modèle permettant la prise en compte du temps de décharge des neurones. Ce type de neurone a ainsi la possibilité de garder une trace de son activité sur une période de temps donnée, ce qui permet, par exemple, l'émergence de corrélations entre les activités de différents neurones du réseau (voir figure 1.4).

Le neurone reçoit en entrée une valeur, I, continue dans [0,1]. Le calcul de son activité est effectué selon les équations :

$$Sortie(t+1) = H(A(t) - 0.5) \tag{1.5}$$

Avec H, la fonction de Heaviside et A le potentiel membranaire du neurone tel que :

$$A(t+1) = f(I).I(t) + (1 - f(I)).A(t) \tag{1.6}$$

$$f(I) = d.(1 - I) + a.I \tag{1.7}$$

a et d étant des constantes.

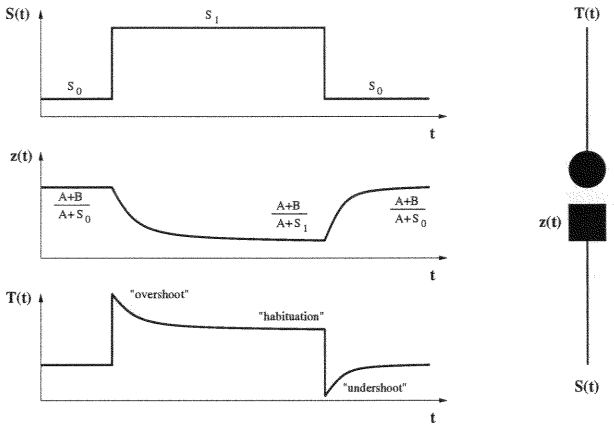


FIG. 1.5 – *Le modèle* gated dipole. *Le signal post-synaptique T(t) varie en fonction du temps, du signal pré-synaptique S(t) et du coefficient de transmission de la synapse z(t). D'après [Grossberg, 1984]*

1.2.5 Le dipole

Il ne s'agit plus ici de modéliser le neurone, mais de descendre à un niveau plus fin dans la transmission de l'information biologique en modélisant le rôle de la synapse. Dans [Grossberg, 1984], Grossberg propose son modèle de *gated dipôle*, permettant de définir l'évolution du signal de sortie d'une synapse au cours du temps.

Le principe Comme le montre la figure 1.5, le modèle de Grossberg simule le signal de sortie, *post-synaptique*, de la synapse $T(t)$, à partir du signal d'entrée, *pré-synaptique*, $S(t)$ et le *coefficient de transmission* de la synapse $z(t)$.

Le signal post-synaptique se détermine par l'équation :

$$T(t) = S(t).z(t) \tag{1.8}$$

tandis que la variation du coefficient de transmission synaptique au cours du temps répond à l'équation :

$$\frac{d}{dt}z(t) = A.(B - z(t)) - S(t).z(t) \tag{1.9}$$

Les variables A et B de la synapse représentent respectivement la vitesse de régénération et la régénération maximale de la synapse.

1.2.6 La colonne corticale

Au delà des modélisations du neurone, il existe différents degrés de modélisation des paradigmes neuronaux. Certain travaux descendent plus bas dans la précision des modèles (les modèles à compartiment). D'autres simulent ces paradigmes à un niveau de granularité plus élevé.

Il existe ainsi des unités de base plus complexes dans les modèles neuromimétiques. Par exemple notre équipe travaille sur des modèles de réseaux inspirés de la biologie, construits à partir de *colonnes corticales*.

Issue de nombreux travaux menés sur le fonctionnement du cortex [Hubel et Wiesel, 1977; Mountcastle, 1978], *la colonne corticale* symbolise l'organisation verticale des neurones dans l'épaisseur du cortex. Une colonne corticale représente un ensemble d'une centaine de neurones biologiques organisés de manière similaire quelle que soit la zone du cortex. Comme le cortex en général, une colonne corticale, qui prend l'épaisseur du cortex, est organisée en six couches, chacune d'entre elles ayant un rôle précis dans les relations de la colonne avec ses nombreuses consœurs. Les trois premières couches communiquent avec des colonnes situées à l'intérieur du cortex. Les neurones de la couche quatre reçoivent des informations de l'extérieur du cortex. Ceux des deux dernières couches envoient des informations à l'extérieur du cortex.

Une colonne corticale est constituée de différents types de neurones, jouant chacun un rôle particulier dans l'activité propre de la colonne corticale. Il est ainsi possible de trouver, entre autres types, des neurones inhibiteurs et des neurones excitateurs. Chaque type de neurones a sa propre connectivité et interagit avec différents types de neurotransmetteurs.

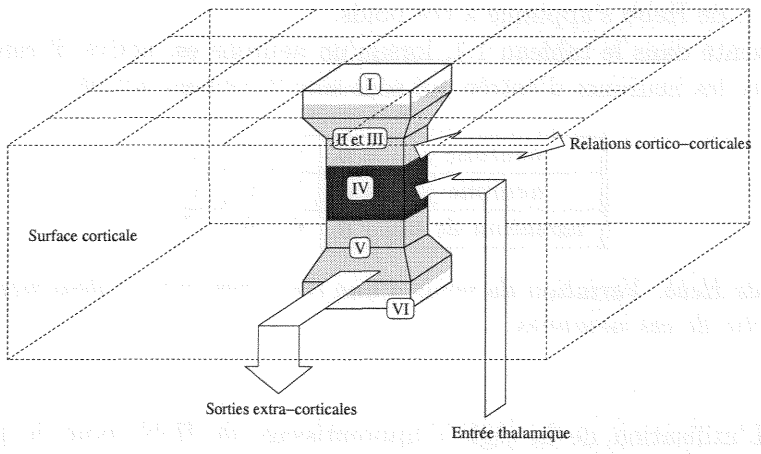


FIG. 1.6 – *Les différentes connexions au sein d'une colonne corticale.*

A partir de ces constations biologiques et des travaux de Burnod [Burnod, 1989], Alexandre et Guyot ont modélisé un modèle informatique de la colonne corticale [Alexandre, 1990; Guyot, 1990] qui a ensuite permis de construire des modèles connexionnistes complexes, et même d'autres unités de base, les *maxi-colonnes*, regroupement topologique de colonnes. Le plus souvent la colonne corticale est simulée en un seul objet reproduisant certaines propriétés des colonnes biologiques. En effet, la simulation des différents neurones, et de leur connectivité, présents dans une colonne serait complexe en terme d'implantation, et coûteuse en temps d'exécution.

1.3 L'apprentissage au sein d'un neurone artificiel

L'apprentissage est l'une des propriétés fondamentales des réseaux de neurones. Il permet au réseau de se spécialiser sur un problème spécifique à partir de son expérience. L'apprentissage d'un réseau connexionniste entraîne, localement, un apprentissage des neurones qui le composent. Cet apprentissage permet de configurer différentes variables du neurone, les poids le plus souvent, afin de spécialiser le traitement de ses entrées.

1.3.1 La règle de Hebb

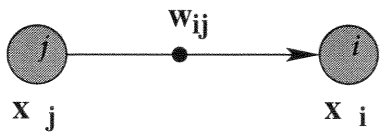


FIG. 1.7 – *Le poids w_{ij} est associé à la connexion entre les neurones i et j*

La règle d'apprentissage de Hebb [Hebb, 1949] est une règle de renforcement des corrélations entre neurones binaires.

Principe Comme rappelé dans la figure 1.7, un neurone associe un poids à chacune de ses entrées, ces entrées étant des liens avec d'autres neurones. La règle d'apprentissage, *non supervisée* (voir section 2.1.1), de Hebb s'applique à ces poids.

Comme représenté dans le tableau 1.1, lorsqu'un neurone est activé, il renforce les poids de ses connexions avec les neurones d'entrée qui sont simultanément activés.

neurone i	0	0	1	1
neurone j	0	1	0	1
variation de w_{ij}	**0**	**0**	**0**	↗

TAB. 1.1 – *Règle de Hebb. Variation du poids d'une connexion reliant deux neurones en fonction des valeurs de sortie de ces neurones.*

Modélisation L'utilisation de la règle d'apprentissage de Hebb pour le poids w_{ij} peut se modéliser comme suit :

$$w_{ij}^{(t)} = w_{ij}^{(t-1)} + \delta.x_i.x_j \tag{1.10}$$

Avec x_i activation du neurone i, δ une valeur positive, et $w_{ij}^{(t)}$ valeur de w_{ij} à l'instant t.

1.3.2 La règle de Widrow-Hoff

La règle d'apprentissage de Widrow-Hoff est une règle de correction des poids permettant de rapprocher l'activation du neurone d'une valeur connue.

Principe Avec cette règle d'apprentissage *supervisée* le neurone connaît la valeur désirée en sortie, et il va chercher à adapter ses poids pour se rapprocher de cette valeur désirée.

La règle d'apprentissage de Windrow-Hoff permet à un neurone de corriger ses différents poids w_{ij}, en suivant l'équation suivante :

$$w_{ij}^{(t)} = w_{ij}^{(t-1)} + \delta.e_i.x_j \tag{1.11}$$

$$e_i = \text{Valeur de sortie de i - Valeur désirée de i} \tag{1.12}$$

Avec x_j activation du neurone j, δ une valeur positive, $w_{ij}^{(t)}$ valeur de w_{ij} à l'instant t et e_i l'erreur commise par le neurone i.

1.4 Conclusion

Les réseaux de neurones sont donc constitués d'unités de base hétérogènes. Il est en effet difficile de trouver un modèle générique de simulation des unités de base des réseaux connexionnistes. Du neurone formel de McCulloch et Pitts aux modèles de colonnes, en passant par les variétés de neurones temporels, seules les notions topologiques d'entrées et de sortie(s) des unités semblent unir ces modèles.

Les fonctions d'activation, la présence et le nombre de variables locales et les règles d'apprentissages sont spécifiques aux modèles et adaptés aux propriétés recherchées des neurones.

Il n'est pas non plus possible de déterminer une granularité dans la simulation des neurones. La recherche avance actuellement à la fois sur des paradigmes de bas niveau, comme le neurone impulsionnel, et sur des paradigmes de haut niveau, comme les colonnes corticales.

Espérer offrir un outil d'aide au développement des réseaux connexionnistes implique de faciliter l'implantation de tous les types de neurones vus précédemment, tout en laissant ouverte la possibilité d'extension de ces modèles. Il faudra aussi tenir compte des connexions entre les neurones, connexions créant les réseaux connexionnistes proprement dits.

Chapitre 2

Du neurone artificiel aux réseaux neuromimétiques

Définition 1 (D'après [Touzet, 1992])
Les réseaux de neurones artificiels sont des réseaux fortement connectés de processeurs élémentaires fonctionnant en parallèle. Chaque processeur élémentaire calcule une sortie unique sur la base des informations qu'il reçoit. Toute structure hiérarchique de réseaux est évidemment un réseau.

Pris isolément un neurone formel ne peut exécuter que des fonctions très simples, des séparations linéaires dans le cas du neurone formel de Mc Culloch et Pitts. Un neurone formel, à l'image de son modèle biologique, ne peut donc pas présenter de comportement intelligent [Durand, 1995]. C'est donc, cette fois encore à l'image du modèle biologique, le nombre et la connectivité des neurones qui leur donnent leurs capacités.

Si un réseau de neurones artificiels est une somme de neurones artificiels connectés, toute somme de neurones admettant une ou plusieurs entrées et sorties et disposant d'une topologie lui permettant de déterminer ces sorties à partir de ces entrées est aussi un réseau connexionniste. Il n'existe aucune limite théorique pour les topologies connexionnistes. Par contre il existe des constantes topologiques, algorithmiques et sémantiques, que nous allons essayer de présenter ici en évoquant les modèles connexionnistes les plus courants. Nous pourrons ainsi en déduire les spécificités propres au domaine connexionniste qu'il nous faudra intégrer, sans toutefois nous y limiter, lors du développement d'un simulateur de réseaux de neurones. Des descriptions plus complètes et théoriques des différentes topologies peuvent être trouvées dans [Hertz *et al.*, 1991; Abdi, 1994; Rumelhart *et al.*, 1986].

2.1 Quelques spécificités des réseaux

Depuis le neurone formel de Mc Culloch et Pitts, les topologies de réseaux de neurones développées furent nombreuses et variées. S'il est dérisoire de vouloir en présenter ici une liste exhaustive, il reste néanmoins possible de tenter de les classifier. Il existe, à cet effet, de nombreuses taxonomies, la figure 2.1 en présente une parmi d'autres. De ces diverses classifications, il est possible d'extraire des redondances dans les topologies et dans les algorithmes associés à celles-ci.

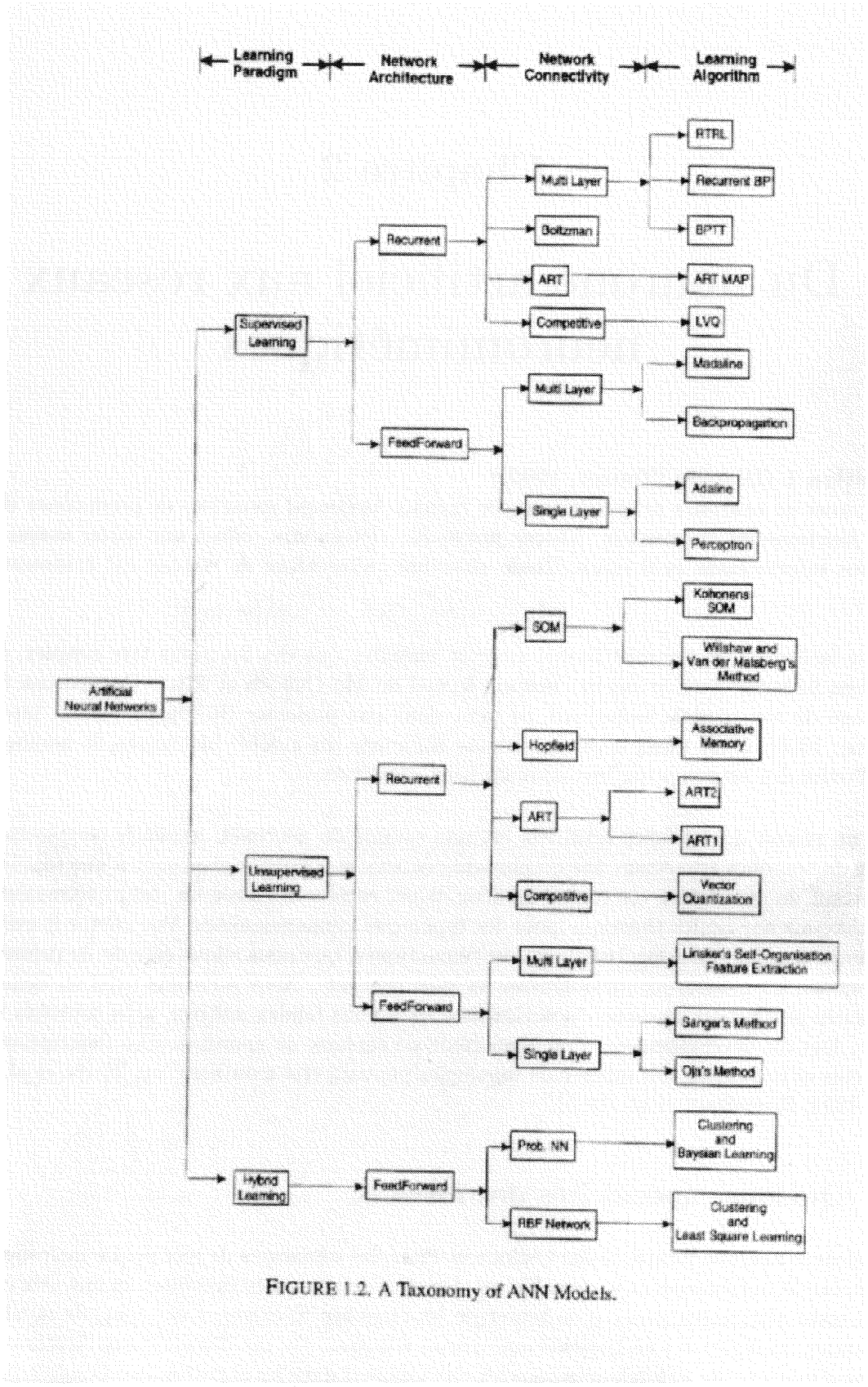


FIGURE 1.2. A Taxonomy of ANN Models.

FIG. 2.1 – *Un exemple de taxonomie des réseaux de neurones. D'après [Sundararajan et Saratchandran, 1998]*

Il est ainsi possible de décrire un réseau en déterminant s'il s'agit d'un :

- Réseau à apprentissage supervisé ou non.
- Réseau feed-forward ou récurrent.
- Réseau à couche(s) ou non.
- Réseau à architecture évolutive.

Nous allons, dans la suite de ce chapitre, définir ces différentes propriétés des réseaux connexionnistes.

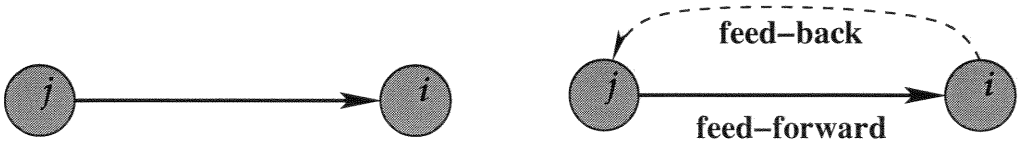
Quelques définitions

Pour faciliter la description des différents modèles connexionnistes, nous définissons ici les termes fréquemment usités.

- Dans un réseau de neurones artificiels, différents types de neurones se distinguent, en fonction de leurs contacts ou non avec le monde extérieur au réseau [Bougrain, 2000]:
  - ◇ *Les neurones d'entrée*
    Ces neurones reçoivent l'information de l'extérieur du réseau et la transmettent aux neurones situés à l'intérieur de celui-ci. Ces neurones n'effectuent généralement aucune autre opération sur les entrées. Ces neurones sont des neurones formels avec une entrée unique, un poids de 1 sur cette entrée et l'*identité* comme fonction de transfert.
  - ◇ *Les neurones de sortie*
    Ces neurones transmettent le résultat du calcul effectué par le réseau vers l'extérieur de celui-ci. Au contraire des neurones d'entrée, ces neurones traitent l'information avant de la transmettre, les poids des connexions ne sont pas toujours à 1 et la fonction de transfert est quelconque.
  - ◇ *Les neurones cachés*
    Ce sont les neurones situés à l'intérieur du réseau et n'ayant aucun lien avec l'extérieur.
- De la même manière, une *couche d'entrée* sera composée exclusivement de neurones d'entrée, une *couche de sortie* seulement de neurones de sortie et une *couche cachée* uniquement de neurones... cachés.
- En terme de connectivité, une couche A est dite *totalement connectée* à une couche B, si chacun des neurones appartenant à la couche A est connecté à tous les neurones de la couche B.
- Il existe deux types principaux de connexions entre deux neurones, les connexions *Feed-Forward* et les connexions *Feed-Back*, selon la réciprocité ou non du flux de l'information au travers de cette connexion, comme montré dans la figure 2.2. Une connexion *Feed-Forward* véhicule l'information principale d'un neurone vers un autre dans le sens de flux du réseau (voir figure 2.2(a)). La connexion *Feed-Back*, connexion retour, renvoie au neurone la réaction du neurone récepteur (voir figure 2.2(b)).
  Ces différents types de liens signifient que le neurone peut *typer ses liens* et ne pas réagir de manière identique à un stimuli selon le type de la connexion.

2.1.1 L'apprentissage

Nous abordons, dans cette partie, une caractéristique primordiale des réseaux connexionnistes, leur calibrage par apprentissage. Si les réseaux connexionnistes restent une grossière simu-



(a) Connexion *Feed-Forward*. Le neurone i peut lire la sortie du neurone j. Le neurone j n'a aucune information de la part du neurone i.

(b) Connexion *Feed-Back*. Elle permet au neurone j de recevoir de l'information en retour du neurone i.

FIG. 2.2 – *Les deux types de connexions du neurone i au neurone j.*

lation de leurs modèles d'inspiration biologique, les algorithmes et les capacités d'apprentissage de ces modèles artificiels sont encore bien plus éloignés des potentiels et des paradigmes des réseaux de neurones naturels.

Les réseaux de neurones artificiels sont construits de manière à remplir une tâche bien définie. Un réseau de neurones doit, à partir de l'information reçue par l'intermédiaire de ses neurones d'entrée, la traiter de manière à transmettre, par ses neurones de sortie, le résultat attendu.

Par exemple, un réseau construit pour la reconnaissance de chiffres manuscrits doit pouvoir recevoir, en entrée, une image, et indiquer, en sortie, lequel des chiffres est représenté sur cette image (voir figure 2.3).

Comme dans le cerveau, pour lequel l'intelligence, ou du moins les capacités cognitives, n'apparaissent qu'avec l'expérience et le savoir, un apprentissage est indispensable pour obtenir des résultats pertinents d'un réseau de neurones artificiels. Il est ainsi nécessaire de déterminer la topologie adaptée au problème posé et les poids des différentes connexions permettant d'effectuer la tâche attendue. Le but de l'apprentissage est, le plus souvent, la détermination des poids du réseau connexionniste et, plus rarement, l'obtention d'une topologie adéquate.

Il existe donc deux phases primordiales dans la vie des principaux réseaux de neurones, qui sont, dans l'ordre temporel, la *phase d'apprentissage* pendant laquelle le réseau spécifie ses paramètres, et la *phase d'exploitation*, durant laquelle il traite effectivement des entrées nouvelles.

- La phase d'apprentissage s'exécute à l'aide d'un *corpus* contenant des données répondant aux propriétés attendues du réseau. L'apprentissage doit lui permettre de se paramétrer afin d'atteindre ces propriétés. Ce corpus d'apprentissage est parfois couplé à un second corpus, de *test*, permettant de tester les résultats obtenus par le réseau et la capacité de celui-ci à généraliser les propriétés apprises à des données inconnues.
- La phase d'exploitation est beaucoup plus simple. Le réseau est paramétré et devient une boite noire traitant les données présentées en entrée suivant le mode opératoire préalablement appris.

Certaines architectures, généralement d'inspiration directement biologique, ne connaissent pas ces deux phases. Ces architectures sont, comme le cerveau, continuellement en apprentissage.

Il existe deux grandes catégories dans les méthodes d'apprentissage des réseaux connexionnistes, selon que l'apprentissage est ou non *supervisé*.

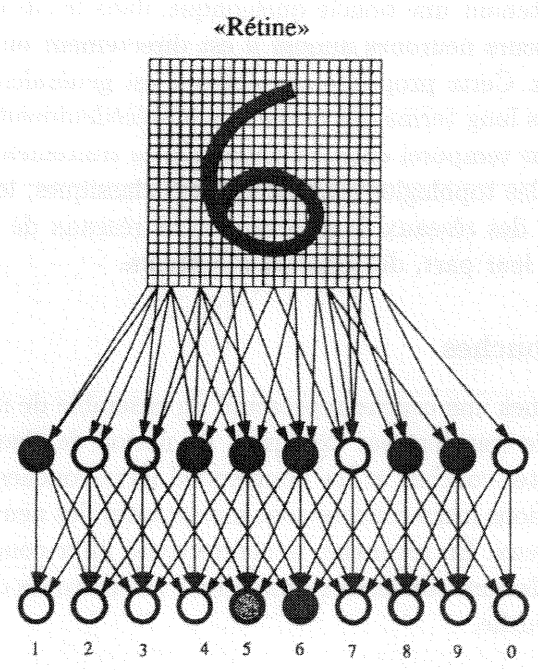


FIG. 2.3 – *Après apprentissage le réseau a la capacité de reconnaître des chiffres manuscrits.*

L'apprentissage supervisé

Un apprentissage est supervisé quand chaque exemple du corpus d'apprentissage est couplé à la solution attendue. Ce résultat permet au réseau de connaître son erreur sur chacun des exemples, et d'adapter ainsi ses différents paramètres afin de se rapprocher du résultat souhaité.

L'apprentissage non supervisé

Un apprentissage est non supervisé quand le réseau ne dispose pas de la solution attendue sur les exemples pour l'aider à ajuster ses paramètres. Ce cas s'applique essentiellement à des problèmes pour lesquels il n'existe pas de réponses définies aux exemples présentés. Le réseau cherche, pour ce type d'apprentissage, à représenter au mieux l'espace des exemples qui lui sont présentés. La solution ne lui étant pas donnée, charge est au réseau de découvrir les régularités de son corpus d'apprentissage.

2.1.2 Réseau feed-forward ou réseau récurrent

Les réseaux *feed-forward*, dits aussi non bouclés, sont constitués de neurones dont la sortie n'influence jamais les entrées. Le calcul effectué par le réseau peut être vu comme un flux transitant des entrées vers les sorties en un parcours unique et unidirectionnel à travers les différents neurones.

Au contraire, les réseaux *récurrents*, ou bouclés, sont des réseaux contenant des neurones pour lesquels un résultat précédemment calculé et transmis en sortie peut influencer ses entrées. Ceci peut être le résultat d'un neurone bouclant sur lui-même (sa sortie constitue l'une de ses

entrées), ou d'un réseau contenant une boucle quelconque, dans le cas où la sortie d'un neurone est une entrée d'un ou plusieurs neurones auquel il est directement ou indirectement connecté, par des connexions *feed-back*. Cette propriété topologique est généralement utilisée pour garder une mémoire à plus ou moins long terme des événements précédemment rencontrés par le réseau ou pour introduire un facteur temporel dans les capacités de traitement du réseau.

A titre d'exemple, dans les topologies connexionnistes classiques, les perceptrons et les perceptrons multi-couches sont des réseaux feed-forward. Les réseaux de Hopfield, les modèles de Elman et Jordan sont, pour leur part, des réseaux récurrents.

2.1.3 Les réseaux à couches

Dans un réseau de neurones une *couche* représente un ensemble de neurones ayant les mêmes propriétés (même fonction de transfert par exemple). La notion de réseaux à couches permet de distinguer les connexions inter-couches et les connexions intra-couches d'un neurone [Jodouin, 1994]. En général, les connexions intra-couches sont inexistantes, les neurones d'une même couche ne sont pas connectés entre eux. En revanche, la connectivité inter-couches laisse tous les choix. Deux couches sont dites *totalement connectées* quand tous les neurones d'une couche sont connectés à tous ceux de l'autre couche.

Un point intéressant dans le traitement des réseaux à couches réside dans la synchronisation de ces modèles, c'est un point sur lequel nous reviendrons souvent dans ce manuscrit. Au niveau de l'exécution, les neurones d'une même couche peuvent être considérés comme synchrones, ils reçoivent leurs entrées simultanément et le calcul de leur activation peut se faire en même temps.

Dans un réseau à couches les calculs se font par *flux*, c'est à dire que l'information est transmise de couche en couche, les différentes couches étant évaluées consécutivement, suivant les connexions entre celles-ci.

A titre d'exemple, et toujours dans les topologies connexionnistes classiques, les perceptrons, les modèles de Elman et Jordan sont des réseaux à couches tandis que le modèle de Hopfield n'en possède pas.

2.1.4 Les réseaux évolutifs

Nous ne parlons pas ici exclusivement d'une propriété topologique des réseaux de neurones, mais d'une propriété portant sur les algorithmes d'apprentissage de ces réseaux ayant des conséquences sur leur architecture. Les réseaux *évolutifs* représentent en effet la famille des réseaux connexionnistes dont la topologie évolue au cours de l'apprentissage.

La phase d'apprentissage d'un réseau évolutif a pour but de configurer le réseau à partir de son corpus d'exemples. Le réseau doit ainsi extraire des régularités du corpus d'exemples et généraliser ces régularités aux données qui lui seront présentées en phase d'exécution.Pour obtenir un bon apprentissage, c'est à dire un bon traitement des données futures, les nombres de neurones et de connexions du réseau sont des facteurs primordiaux. Ils influencent notamment la capacité du réseau à éviter les minima locaux et l'apprentissage dit *'par cœur'* (le réseau ne répond correctement que sur les données de son corpus d'apprentissage). Mais, si la détermination de la topologie connexionniste est primordiale, il n'existe que peu de méthodes formelles pour le faire. Comme nous le verrons plus loin avec les perceptrons multi-couches, section 2.2.2, la théorie peut donner le nombre de couches du réseau, mais ni le nombre de neurones par couches, ni les connexions liant les neurones des différentes couches.

La famille des réseaux évolutifs cherche à résoudre ce problème en intégrant à la phase d'apprentissage la détermination dynamique de la topologie des réseaux.

Pour cette recherche dynamique de topologie, il existe deux politiques différentes : la construction dynamique du réseau et la correction d'un réseau disposant d'une topologie en début d'apprentissage.

La première méthode, dite *incrémentale*, consiste, à partir d'une topologie minimale, à ajouter des neurones et des connexions au réseau, au cours de l'apprentissage, afin d'améliorer les performances de celui-ci.

Dans la seconde méthode, dite par *élagage*, une architecture neuronale complète est présente au démarrage de l'apprentissage. Ici l'algorithme d'apprentissage effectue, en plus de la traditionnelle correction des poids associés aux connexions, la suppression de connexions et de neurones afin d'optimiser les performances du réseau [LeCun *et al.*, 1990; Reed, 1993].

Les réseaux évolutifs, par leur adaptation topologique au corpus d'exemples pourraient apparaître comme le meilleur moyen d'obtenir la topologie optimale. Mais ils ne sont pas à l'abri des problèmes de performances, comme le montrent Hérault et Jutten dans [Hérault et Jutten, 1994] au dépend des algorithmes incrémentaux *Grow and learn* [Alpaydin, 1990] et RCE [Reilly *et al.*, 1987].

Les réseaux de type Cascade Correlation, Growing Neural Gas sont des exemples classiques de réseaux connexionnistes incrémentaux.

2.2 Quelques architectures connexionnistes classiques

Cette partie est consacrée à une présentation des modèles connexionnistes incontournables. Ces modèles reflètent les différentes topologies des réseaux de neurones dans le sens où la grande majorité des réseaux classiques et expérimentaux sont dérivés de ces modèles ou utilisent leurs paradigmes.

Ainsi un simulateur de réseau de neurones devra, au minimum, permettre d'implanter ces différents modèles voire, à défaut, pouvoir justifier les choix entraînant cette incapacité.

2.2.1 Le perceptron

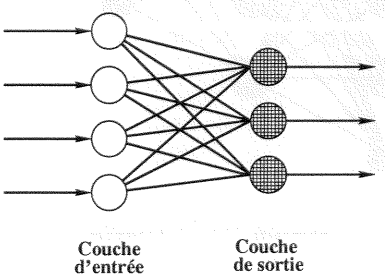


FIG. 2.4 – *Un exemple de perceptron simple.*

L'architecture Développé par Rosenblatt dans les années 50, le Perceptron, appelé aussi *perceptron simple* est historiquement le premier réseau de neurones artificiels. Il se compose de deux couches de neurones, neurones de type Mc Culloch et Pitts. La couche d'entrée, couramment

appelée *rétine*, reçoit l'information. La seconde couche est la couche de sortie, elle transmet le résultat du traitement des entrées par le perceptron.

La couche de sortie est totalement connectée à la couche d'entrée, (voir figure 2.4). Dans ce modèle les valeurs d'entrées comme les valeurs de sortie sont binaires (0 ou 1).

L'apprentissage Le perceptron utilise un apprentissage supervisé. Cet apprentissage ne concerne que les neurones de la couche de sortie, qui doivent ajuster les valeurs des poids associés à leurs entrées.

Les méthodes utilisées sont donc les méthodes d'apprentissage du neurone binaire de Mc Culloch et Pitts, la règle *Delta* ou *Widrow-Hoff* [Widrow et Hoff, 1960]. Rosenblatt a démontré que, si une solution existe, le perceptron la trouvera après un nombre fini d'itérations.

Propriétés du modèle Les perceptrons ont la capacité de résoudre des problèmes de type *linéairement séparable*. L'exemple le plus souvent employé montre qu'un perceptron peut résoudre les fonctions ***et logique*** et ***ou logique***, mais ne peut pas résoudre le ***xor***, qui n'est pas linéairement séparable.

2.2.2 Le perceptron multi-couches

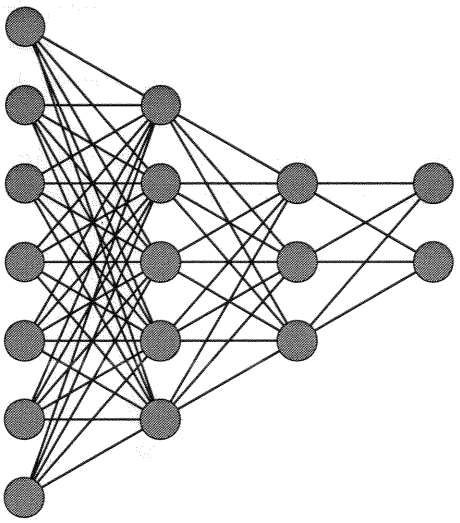


FIG. 2.5 – *Un exemple de perceptron multi-couches contenant deux couches cachées.*

L'apparition de l'algorithme de *rétro-propagation du gradient* a permis de fortement étendre le champ des compétences du connexionnisme, en lui permettant de résoudre des problèmes autres que linéairement séparables. L'apport de cet algorithme d'apprentissage est de fournir des règles permettant l'apprentissage de couches cachées ajoutées aux perceptrons classiques, couches placées entre la couche d'entrée et la couche de sortie (voir figure 2.5). Un perceptron multi-couches classique est donc un perceptron simple, auquel est ajouté un nombre fini de couches cachées, chaque couche étant, le plus souvent, totalement connectée à la précédente.

Propriétés du modèle

Définition 2
Les réseaux de neurones à trois couches sont capables d'approximer l'ensemble des fonctions continues de $\mathbb{R}^p$ ou de compacts de $\mathbb{R}^p$ dans $[0,1]^n$.

Le perceptron multi-couches est considéré comme un approximateur universel de fonctions. Il permet en effet de simuler toutes les fonctions continues de $\mathbb{R}^p$ dans $\mathbb{R}$.

Il existe de nombreuses démonstrations de cette assertion [Hecht-Nielsen, 1989; Hornik *et al.*, 1989] qui fait d'un tel réseau connexionniste à trois couches (une couche d'entrée, une couche cachée et la couche de sortie) un *approximateur universel.* Mais, s'il est fait référence à l'existence du réseau, il n'est pas fait allusion à la topologie exacte du réseau permettant d'approximer une fonction donnée. Certaines solutions théoriques nécessitent une couche cachée contenant un nombre de neurones infini tandis qu'elles s'approximent parfaitement à partir d'un réseau à quatre couches [Chester, 1990]. De plus, un trop grand nombre de neurones limite les capacités de généralisation d'un réseau.

L'apprentissage Le mode d'apprentissage le plus usuel du perceptron multi-couches est un apprentissage supervisé, suivant l'algorithme de rétro-propagation du gradient [Rumelhart *et al.*, 1986; le Cun, 1985; le Cun, 1987]. Cet algorithme permet de transmettre à tous les neurones des couches cachées leur part dans l'erreur totale faite par le réseau à la présentation d'un exemple. Les neurones peuvent ainsi tenir compte de leur erreur pour corriger leurs poids.

Cet apprentissage nécessite, pour chaque élément du corpus d'exemples, un passage par chacun des neurones de chacune des couches afin de calculer la sortie proposée par le réseau. Puis, après détermination de l'erreur globale commise par l'ensemble du réseau, un second passage de la couche de sortie vers la couche d'entrée est réalisé afin de transmettre à chacun des neurones sa part dans l'erreur globale pour qu'il puisse ajuster les poids de ses connexions.

Il est, de plus, nécessaire de présenter un nombre d'exemples conséquent pour l'apprentissage, nombre croissant avec la complexité du réseau.

La convergence d'un perceptron multi-couches nécessitant, pour finir, de nombreuses itérations, l'apprentissage de cette architecture est très lourd en terme de calcul, et par là même, très lent.

Si la structure à couches des perceptrons trouve une justification biologique, notamment dans les structures en couche du cortex, la réalité biologique de son algorithme d'apprentissage est beaucoup plus discutée [Zipser et Andersen, 1988; Alexandre et Guyot, 1995; Guigon, 1993]. L'algorithme de rétro-propagation du gradient est essentiellement une solution purement mathématique au problème posé par l'apprentissage des réseaux à couches. Il s'appuie sur la rétro-propagation de l'erreur commise par le réseau, et donc sur un retour d'information à travers une connexion *feed-back.* Une simulation biologiquement plausible de cette rétro-propagation impliquerait la présence de connexions et de neurones *feed-back*, présence qui compliquerait fortement la topologie des perceptrons.

Le perceptron multi-couches correspond actuellement à l'architecture la plus populaire et la plus répandue dans le monde connexionniste, pour les applications des réseaux de neurones comme pour la recherche.

Des recherches sur les algorithmes d'apprentissage sont effectuées dans le but de réduire le temps nécessaire à cet apprentissage. Les résultats obtenus peuvent aller de la mise en place d'algorithmes alternatifs [Fahlman, 1989], au développement de nouveaux paradigmes, comme les

SVM[2] [Cortes et Vapnik, 1995; Boser *et al.*, 1992], architecture considérée comme une démarche à part, à la limite du connexionnisme, qui connaît néanmoins un certain engouement dû à de très bonnes performances.

Il existe aussi des recherches portant sur la topologie, pratiquées afin d'optimiser les performances connues des perceptrons multi-couches et d'étendre le champ des propriétés du modèle. Des topologies alternatives à celle du perceptron multi-couches sont ainsi apparues, introduisant les aspects évolutifs ou récurrent.

2.2.3 Les réseaux à couches récurrents

Pour introduire une dimension temporelle aux perceptrons multi-couches et pour leur donner une capacité de mémoire à court terme, la notion de récurrence a été introduite dans le modèle du perceptron multi-couches (il existe des méthodes alternatives, non récurrentes, de prise en compte du temps dans les réseaux à couches, le réseau TDNN [Waibel *et al.*, 1989], utilisé pour la reconnaissance vocale, en est un exemple illustre).

Parmi les nombreuses architectures de réseaux récurrents à couche, nous présentons ici deux modèles classiques, ceux de Jordan et d'Elman.

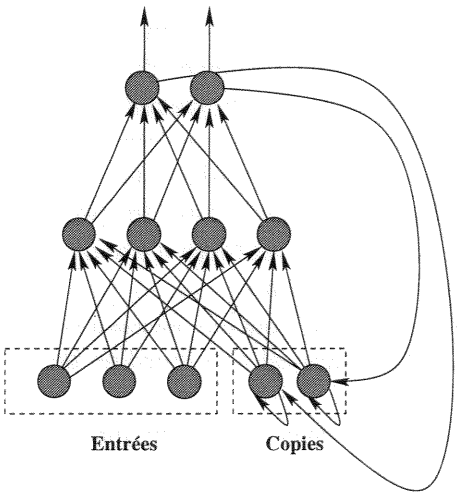


FIG. 2.6 – *Modèle de Jordan. La couche de sortie est recopiée et intégrée dans la couche d'entrée, par les liens* feed-back *du réseau.*

• Le modèle de Jordan

L'architecture Dans ce modèle, décrit dans [Jordan, 1986] et ayant pour base topologique un perceptron multi-couches à une couche cachée, la récurrence permet d'introduire une mémoire des sorties du réseau. La sortie du réseau est intégralement copiée dans une couche, que nous appellerons *copie*, qui est incluse dans la couche d'entrée du réseau. Les poids entre la couche de sortie et la couche de copie sont fixés à 1 et ne sont pas modifiés par l'apprentissage. Les

2. Support Vector Machines

autres poids sont corrigés par apprentissage, effectué à l'aide d'un des algorithmes classiques de rétro-propagation du gradient. Pour finir, chaque neurone de la couche de copie possède un lien récurrent sur lui-même.

Les propriétés Ainsi configuré, ce type de réseau permet de calculer des fonctions de type :

$$x_t = f(x_{t-1}, e_t) \tag{2.1}$$

où x_t représente la sortie du réseau à l'instant t et e_t l'entrée courante du réseau en cet instant t.

La mémoire engendrée par les liens de récurrence permet de faire un lien entre les entrées successives en tenant compte de l'ordre de présentation de celles-ci. Il peut permettre, par exemple, de traiter les entrées en séquences.

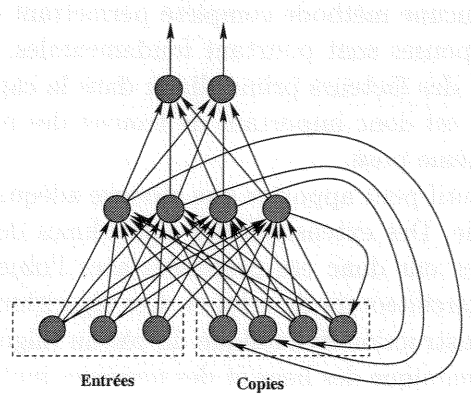


FIG. 2.7 – *Modèle de Elman. La couche cachée est recopiée et intégrée dans la couche d'entrée.*

- **Le modèle de Elman [Elman, 1990]**

L'architecture Ce modèle, comme celui de Jordan, est une extension d'un perceptron multi-couches à une couche cachée. D'un point de vue topologique, c'est cette fois la couche cachée du réseau qui est mémorisée dans la couche de copie, elle-même toujours ajoutée à la couche d'entrée, comme décrit dans la figure 2.7. Les poids des connexions entre la couche cachée et la couche de copie sont toujours fixés à 1 et ne sont pas modifiés par l'apprentissage, ce dernier étant ici aussi une rétro-propagation du gradient de l'erreur.

Les propriétés Cette fois la récurrence dans la topologie apporte au perceptron une prise en compte du contexte dans la détermination de la sortie. Ce type de réseau permet l'approximation d'équations de type [Durand, 1995]:

$$x_t = f(x_{t-m}, \ldots, x_{t-1}, e_t) \tag{2.2}$$

Avec x_t et e_t qui représentent respectivement la sortie et l'entrée de l'exemple t.

Cette topologie est utilisée, par exemple, pour effectuer de l'apprentissage de grammaires.

Il existe de nombreux autres modèles de réseaux de neurones à couches récurrents, certains, contrairement aux architectures d'Elman et de Jordan, associant les poids des connexions récurrentes à l'apprentissage [Gas, 1994].

2.2.4 Les réseaux à couches évolutifs

Comme nous l'avons vu précédemment, un réseau de neurones à trois couches peut être considéré comme un approximateur universel de fonctions continues sur l'espace des réels. L'existence du réseau étant ainsi connue, il reste à construire l'architecture du réseau et, pour ce faire, à répondre aux questions suivantes :

- Combien de neurones doit contenir chaque couche?
- Quels sont les liens reliant les neurones des différentes couches?

Il n'existe actuellement aucune méthode complète permettant de répondre de manière théorique à ces questions. Les réponses sont pourtant fondamentales, le nombre de neurones et de connexions d'un réseau étant des facteurs primordiaux dans la capacité d'apprentissage puis de généralisation d'un réseau. Il est donc important de trouver des politiques de détermination de topologie répondant au problème posé.

Le concept de réseau évolutif peut apporter une réponse adéquate pour résoudre ce problème de détermination de topologie. Des extensions des algorithmes de rétro-propagation classiques des perceptrons multi-couches ont donc été proposées avec l'objectif d'apprendre à la fois les poids du réseau, mais aussi l'architecture de celui-ci. Ces algorithmes peuvent se diviser en deux classes, les algorithmes de construction dynamique du réseau (algorithmes incrémentaux), et les algorithmes d'élimination dynamique des liens et des neurones inutiles au problème (algorithmes d'élagage).

L'élagage est une méthode de recherche d'une topologie optimisée d'un réseau connexionniste à couches. Pour obtenir cette topologie, le principe de ces algorithmes est de partir d'un réseau sur-dimensionné, puis de simplifier la topologie de ce réseau en cours d'apprentissage. Cette simplification se fait en supprimant les poids jugés *peu pertinents*, c'est-à-dire les poids n'ayant qu'une faible, et parfois une mauvaise, incidence sur le résultat attendu du réseau. La suppression des poids entraîne la suppression de la connexion à laquelle ce poids est dédié, ce qui, à force de suppression, optimise la topologie du réseau, et allège son coût, tant au niveau de l'espace mémoire occupé que des temps de calculs nécessaires au réseau lors de ses phases d'utilisation.

Le problème de l'élagage réside donc dans le critère de sélection des poids à éliminer. Si de nombreuses méthodes ont été proposées pour effectuer ce choix [Reed, 1993], deux d'entre elles se retrouvent fréquemment. La première, notamment utilisée par [LeCun *et al.*, 1990; Hassibi et Stork, 1993; Karnin, 1990] se base sur un *calcul de sensibilité de l'erreur* entraînée par la suppression éventuelle de chacun des poids du réseau. La seconde, considérée comme moins performante par [Hérault et Jutten, 1994; Crespo et Mora, 1993], utilise une technique dite d'élagage par *pénalisation de la complexité* [Williams, 1994; Chauvin, 1990].

Dans la plupart de ces méthodes les phases d'apprentissage alternent avec les phases d'élagage, jusqu'à obtention d'une topologie satisfaisante. L'apprentissage à l'aide de ces algorithmes d'élagages est donc souvent lourd en temps de calcul.

Les modèles à couches incrémentaux La méthode, cette fois, consiste à démarrer l'apprentissage avec une architecture minimale, puis à développer l'architecture en ajoutant connexions

et neurones pendant l'apprentissage. Nous présentons, dans la suite, un exemple d'algorithme incrémental de construction de réseau de neurones multi-couches, *Cascade Correlation.*

• **Cascade Correlation**

Cascade Correlation ne correspond pas à proprement parler à une topologie connexionniste. Il s'agit plutôt d'un méta-algorithme de construction, à partir d'un ensemble d'apprentissage, d'un réseau multi-couches d'architecture relativement classique [Zell et al, 1993].

Algorithme Le but de cette méthode est la recherche d'une architecture proche de l'architecture minimale apte à résoudre le problème posé, à approximer l'équation recherchée.

Le squelette de base de cet algorithme est le suivant [Fahlman et Lebiere, 1990; Hoehfeld et Fahlman, 1992] :

- Le réseau démarre avec uniquement sa couche d'entrée et sa couche de sortie, cette dernière étant totalement connectée à la couche d'entrée.
- Les connexions reliant une sortie sont apprises jusqu'à ce que l'erreur du réseau cesse de décroître.
- A l'ajout d'une couche cachée (en fait un unique neurone), celle-ci reçoit en entrée la couche d'entrée et les autres couches cachées préalablement incluses. Elle n'est, pour commencer, pas reliée à la couche de sortie. Les poids de ces nouvelles connexions sont ensuite appris de manière à maximiser leur covariance avec l'erreur commise par le réseau. Par la suite ces poids sont gelés et l'apprentissage ne portera que sur les liens de cette couche cachée à la couche de sortie, à laquelle elle est maintenant connectée.
- Les couches cachées sont ajoutées une à une tout au long de l'apprentissage jusqu'à l'obtention d'une erreur moyenne inférieure à un seuil préalablement fixé.

Dans cet algorithme, l'apprentissage des différents poids se fait à l'aide de l'algorithme de rétro-propagation classique ou de l'une de ses nombreuses variantes.
La figure 2.8 présente l'état du réseau en cours d'apprentissage, après l'ajout de trois neurones cachés.

Ainsi cet algorithme permet l'obtention d'une topologie adaptée aux caractéristiques extraites de l'ensemble des exemples présentés durant la phase d'apprentissage. Le réseau obtenu est de type feed-forward dans le sens où le flux de l'information chemine de la couche d'entrée du réseau à la couche de sortie de celui-ci sans jamais boucler. Les expériences montrent que cet algorithme est plus rapide à l'apprentissage que les algorithmes classiques.

Il existe, comme toujours, des extensions à cet algorithme de construction dynamique des réseaux connexionnistes.

Pruned Cascade Correlation (cité dans [Zell et al, 1993]) ajoute un élagage du réseau après ajout d'une nouvelle unité, afin d'éliminer les connexions les moins pertinentes. La notion de base de cet élagage est la recherche de la minimisation de l'erreur espérée sur l'ensemble de test associé à l'apprentissage du réseau.

De la même manière que Cascade Correlation permet de construire des réseaux équivalents à des réseaux à couches classiques, il existe une version permettant de construire l'équivalent des réseaux à couches récurrents, *Récurrent Cascade Correlation* [Fahlman, 1991]. Pour obtenir ce résultat, l'algorithme dote chaque nouvelle unité cachée d'un lien récurrent sur elle même.

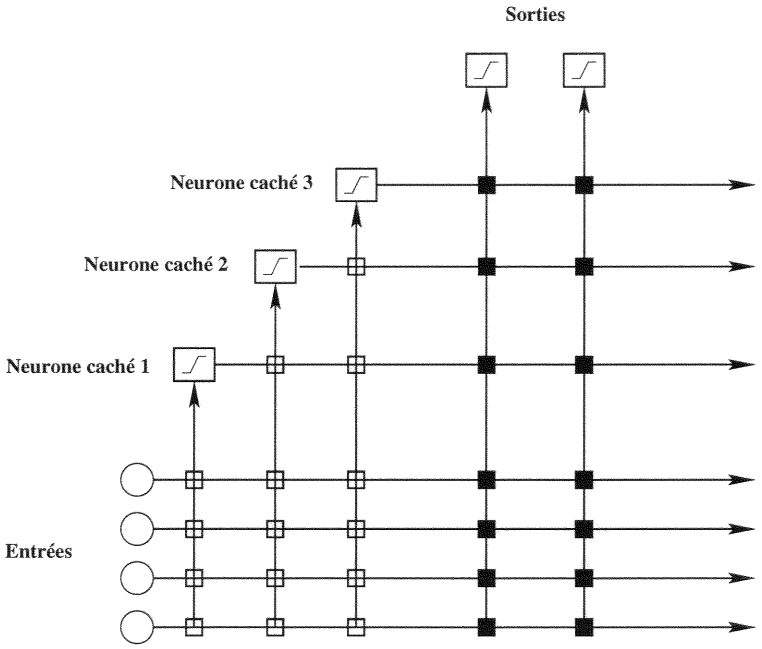


FIG. 2.8 – *Un exemple de réseau construit avec Cascade Correlation après ajout de trois neurones. Les connexions en blanc sont gelées tandis que celles en noir demeurent en état d'apprentissage. D'après [Zell et al, 1993].*

En plus des extensions récurrentes et/ou évolutives des perceptrons multi-couches, il existe de nombreuses autres déclinaisons de cette topologie qui reste l'un des tous premiers modèles de recherche en connexionnisme. Il existe par exemple le modèle OWE [Pican, 1995], une topologie de prise en compte du contexte dans la détermination des poids. Dans ce modèle, que nous détaillerons plus tard dans ce manuscrit, chaque poids du réseau est déterminé par un perceptron multi-couches propre. Cette architecture, coûteuse dans son modèle classique par le nombre de ses neurones et de ses connexions, peut être allégée par l'apport d'un algorithme d'apprentissage incluant un élagage [Bougrain, 2000].

2.2.5 Les réseaux de Hopfield

Les modèles de Hopfield représentent une architecture plus historique que pratique. Ils sont importants car apparus à un tournant de l'histoire du connexionnisme. Ils sont considérés comme la base de son redémarrage [Durand, 1995]. En revanche ils ne sont quasiment plus utilisés dans leur version de base en raison de leur coût en terme de temps de calculs et de leurs relativement faibles performances.

L'architecture Les modèles connexionnistes de Hopfield sont constitués de neurones formels de type Mc Culloch et Pitts, totalement connectés entre eux [Hopfield, 1982]. Tous les neurones de cette architecture sont à la fois neurone d'entrée et neurone de sortie du réseau. La spécificité de ce réseau réside dans une recherche permanente, pour chacun des neurones du réseau, d'un état stable.

Formellement, comme le montre la figure 2.9, un réseau de Hopfield est un réseau récurrent, chacun des neurones du réseau étant connecté à tous les autres, mais pas à lui-même. Les neurones

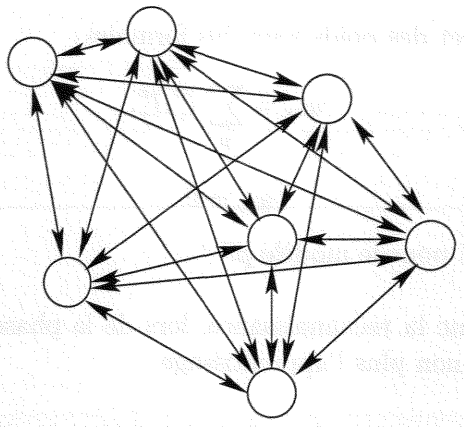


FIG. 2.9 – *Réseau de Hopfield. Un neurone est lié à tous les autres, les liaisons sont symétriques, un neurone n'est pas lié à lui-même et un seul neurone est actualisé par itération. Chaque neurone est à la fois neurone d'entrée et de sortie du réseau.*

disposent de sorties binaires (± 1), et les interconnexions entre les neurones sont symétriques (Pour tous les neurones i et j, $w_{ij} = w_{ji}$).

Champs d'applications Cette architecture est utilisée pour apprendre un certain nombre d'ensembles d'états, ses prototypes, imposés aux neurones. Elle a en effet la capacité de retrouver un ensemble d'états complet à partir d'une partie ou d'une copie bruitée de l'un des prototypes. Par exemple cette architecture a la capacité de reconstruire une image apprise à partir d'un morceau de cette image. Cette architecture connexionniste est ainsi utilisée pour mémoriser des formes et des motifs, lesquels seront reconnus et reconstruits si l'on en présente, après apprentissage, une version partielle ou bruitée au réseau. Ces réseaux sont aussi aptes à résoudre des problèmes d'optimisation pour minimiser une fonction de coût, Hopfield a notamment appliqué son réseau au problème du voyageur du commerce.

L'algorithmique du modèle Après présentation d'une entrée, les neurones sont désignés aléatoirement pour calculer la somme pondérée de leurs entrées, selon l'équation 2.3 et ce jusqu'à l'obtention d'un état stable du réseau. Cette méthode montre la dimension temporelle de cette architecture, en phase d'utilisation, dite de relaxation, l'état du réseau à un instant t dépend directement de son état au temps $t-1$.

$$S_i = \begin{cases} -1 & \text{si } \sum_{j=0}^{n} w_{ij}.x_j + I_i \leq 0. \\ +1 & \text{si } \sum_{j=0}^{n} w_{ij}.x_j + I_i > 0. \end{cases} \tag{2.3}$$

I_i représentant le prototype présenté au neurone i.

L'apprentissage, non supervisé, de cette architecture est assez atypique dans le monde des réseaux de neurones. Il ne repose pas, comme dans la plupart des modèles connexionnistes, sur une convergence elle-même basée sur des essais/erreurs. La connaissance de l'ensemble des états

permet un apprentissage direct des poids selon les formules :

$$w_{ij} = \sum_p I_i^p . I_j^p . \tag{2.4}$$

$$w_{ii} = 0. \tag{2.5}$$

Avec I_i^p état du neurone i pour le modèle p.

Dans ce modèle, c'est donc la reconnaissance, lors de la phase d'*application* du réseau, qui nécessite une convergence et non plus l'apprentissage.

Ainsi défini, un réseau de neurones de ce type peut être considéré comme une mémoire distribuée et associative. Distribuée car l'information est contenue dans l'ensemble des poids du réseau, et associative car c'est l'association de ces poids qui permet de reconstruire les différents ensembles appris.

Les limites Les limites des réseaux de Hopfield sont inhérentes au modèle. Un réseau ne peut stocker qu'un nombre très limité d'états stables, de prototypes. Ce nombre dépend de la taille du réseau et correspond, dans le meilleur des cas, à 15 % du nombre des neurones.

Les extensions Comme pour tous les modèles classiques des réseaux de neurones, de nombreuses modifications de l'architecture et de l'algorithme du modèle de base existent.

Les *réseaux de Hopfield continus* autorisent, pour les neurones, des valeurs de sortie continues dans [-1,+1] en remplaçant la fonction signe, pour la détermination de la sortie des neurones, par une fonction continue, une *tangente hyperbolique* par exemple. Ainsi modifiés les réseaux de Hopfield acquièrent la capacité de résolution de recherche de minima pour des fonctions continues.

Les *réseaux de Hopfield synchrones* [Amit *et al.*, 1985] sont une variante de l'algorithme de relaxation de Hopfield. Ici l'intégralité des neurones est évaluée lors de chaque itération en prenant en compte les sorties des neurones obtenues lors de l'itération précédente. Cette modification algorithmique accélère la convergence du réseau tout en proposant des performances équivalentes, bien que différentes.

La *Machine de Boltzmann* [Hinton et Sejnowski, 1986] est une adaptation du modèle de Hopfield introduisant des neurones cachés et une détermination probabiliste de l'activation. Les machines de Boltzmann permettent principalement de résoudre des problèmes de minima locaux [Abdi, 1994].

2.2.6 Les Cartes auto-organisatrices

Le concept des cartes auto-organisatrices se place à la limite des architectures connexionnistes mathématiques et des modèles biologiques. Si les modèles détaillés précédemment connaissent quelques justifications biologiques, la principale en étant leur élément de base, le neurone formel, ces architectures sont surtout construites pour résoudre certains problèmes par des solutions de calcul distribué. Les cartes auto-organisatrices sont, pour leur part, directement inspirées par les modèles d'auto-organisation constatés dans certaines aires corticales, les aires visuelles par exemple. Ces architectures restent malgré tout adaptées à des problèmes mathématiques très concrets. Elles cherchent généralement à représenter, à partir d'un apprentissage non supervisé, des données complexes et bruitées, et à extraire des régularités d'un ensemble de données.

Les cartes auto-organisatrices de Kohonen

Le plus classique des modèles de réseaux de neurones auto-organisés est sans conteste le modèle de Kohonen [Kohonen, 1989]. Son principe est assez simple. Il s'inspire de constatations neurobiologiques : dans certaines aires sensorielles du cortex, deux stimuli proches activent des neurones proches de l'aire sensorielle, au sens de la topologie de ces aires corticales [Hubel et Wiesel, 1977].

Dans cet esprit, les cartes de Kohonen ont pour objectif la représentation d'un espace d'entrée, représentation comprenant des informations spatiales sur cet espace d'entrée.

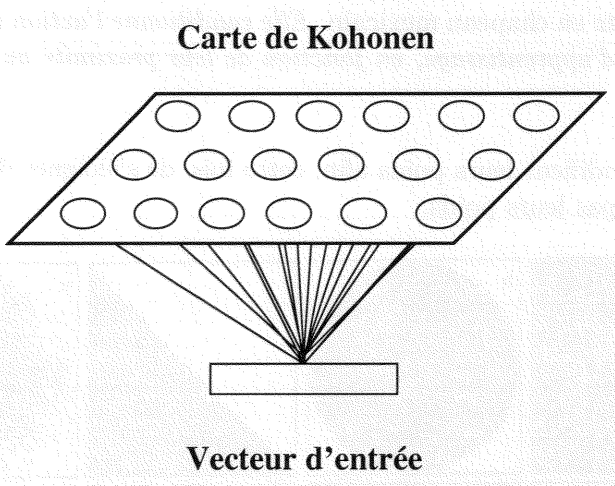


FIG. 2.10 – *Exemple de carte de Kohonen à deux dimensions. La carte dispose d'une topologie de grille et chaque neurone est lié au vecteur d'entrée.*

Architecture Schématiquement, une carte de Kohonen se représente sous forme d'un ensemble de neurones disposés suivant une topologie précise dans un espace. Cet espace est généralement à une ou deux dimensions. Dans ce second cas la carte de Kohonen adopte une représentation en grille, les neurones sont placés dans un plan et chacun d'entre eux est connecté à ses voisins directs.

L'espace d'entrée est représenté sous forme de vecteurs de dimension n. Chaque neurone de la carte admet des poids sous forme d'un vecteur de même dimension, n, et est relié au vecteur d'entrée (voir figure 2.10).

Apprentissage L'apprentissage des poids des neurones d'une carte de Kohonen est un apprentissage non supervisé. Le réseau adapte ses poids aux caractéristiques présentées par l'espace des exemples d'apprentissage. A chaque présentation d'un exemple, les neurones déterminent leur *distance* à celui-ci, calculée à partir d'une métrique préalablement définie, entre le vecteur d'entrée et le vecteur des poids du neurone. Les neurones de la carte mettent ensuite à jour leurs poids selon le principe du *chapeau mexicain* (figure 2.11). Le neurone vainqueur, celui qui est le plus près du vecteur d'entrée au sens de la métrique considérée, modifie son vecteur de poids afin de se rapprocher du vecteur d'entrée. Ensuite, et selon la fonction en chapeau mexicain, les plus proches voisins modifient leurs poids pour se rapprocher plus légèrement du vecteur d'entrée, les

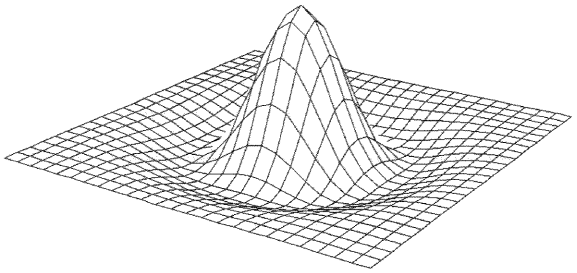


FIG. 2.11 – *Fonction dite en* chapeau mexicain. *Elle conditionne l'action des neurones d'une carte de Kohonen, en phase d'apprentissage, en fonction de leur proximité au neurone vainqueur.*

voisins plus éloignés modifient leurs poids afin, cette fois, de s'éloigner de ce vecteur. Les autres neurones ne modifient pas leurs poids.

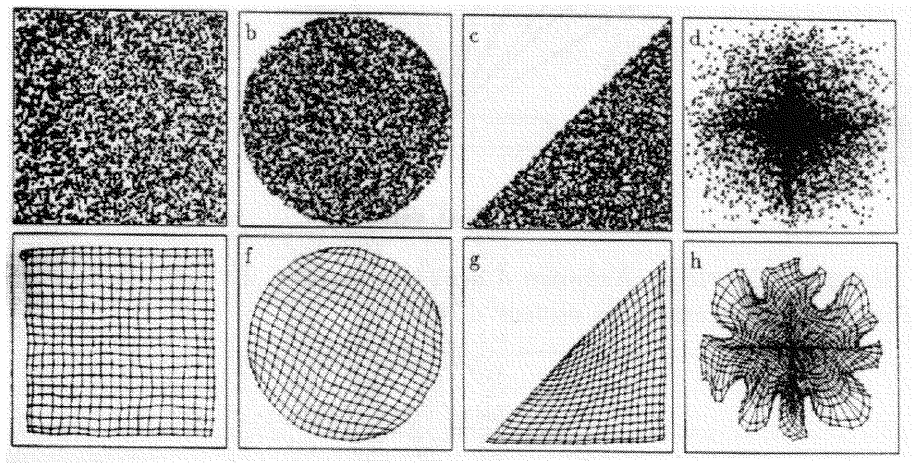


FIG. 2.12 – *Quelques exemples d'apprentissage de réseaux de Kohonen. Les figures a, b, c, d représentent les exemples présentés au réseau et les figures e, f, g, h l'espace des poids du réseau correspondant.*

Par son architecture et son apprentissage, la carte de Kohonen représente, par l'espace des poids de ses neurones, l'espace des exemples présentés à l'apprentissage (voir figure 2.12). Ce modèle permet ainsi d'extraire d'un espace d'exemples des régularités, ou d'effectuer des classifications au sein de cet espace (il crée ainsi une correspondance entre les exemples présentant les mêmes caractéristiques, isolées par le choix de la métrique). Ce réseau permet de représenter un espace de vecteurs de taille n, qui peut être très grand, dans un espace à dimension plus réduite (un ou deux le plus souvent), tout en conservant l'information topologique, de *proximité*, de ces données.

Applications Par leurs capacités de classification et d'extraction de caractéristiques avec une importante composante topologique, les cartes de Kohonen peuvent être utilisées pour compresser des données (des images par exemple) ou comme mémoire associative. Ces cartes sont aussi utilisées dans le but de résoudre des problèmes d'optimisation (le voyageur de commerce par

exemple [Fort, 1988]) ou pour effectuer de l'analyse de données.

Extensions Les cartes auto-organisatrices de Kohonen, dont le principe simplifié est expliqué ci-dessus, ont connu de nombreuses adaptations et variantes pour être adaptées à de nouveaux problèmes, comme la représentation de données non linéaires, ou pour améliorer leurs performances. Ce modèle est aussi fréquemment couplé à une seconde couche connexionniste, pour effectuer de l'aide à la décision.

Bien que les capacités des cartes de Kohonen soient intrinsèquement liées à leur apprentissage non supervisé, qui permet cette "extraction automatique de caractéristiques", il existe des versions supervisées, LVQ[3] [Kohonen, 1989], permettant d'affiner ou d'imposer la classification effectuée par les cartes.

Le modèle neural gas

Ce modèle peut être vu comme un dérivé des cartes de Kohonen. De la même façon que dans ce dernier modèle, un neurone est désigné comme *vainqueur* s'il est le plus près, au sens d'une métrique définie, du vecteur présenté en entrée. La modification des poids, en phase d'apprentissage, est aussi identique aux cartes de Kohonen. C'est la notion de topologie qui distingue ces deux modèles. En effet, quand les cartes de Kohonen disposent d'une topologie fixe, fixée préalablement à l'apprentissage, et cherchent à représenter l'espace des entrées en fonction de celle-ci, les réseaux neural gas [Martinetz et Schulten, 1991; Martinetz et Schulten, 1994] extraient leur topologie, au cours de l'apprentissage, de l'ensemble des exemples.

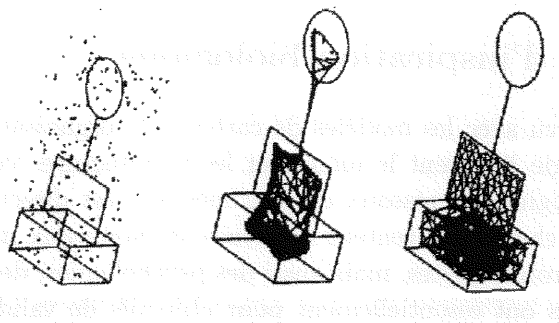


FIG. 2.13 – *Un exemple d'apprentissage avec neural gas. Le réseau doit apprendre un ensemble constitué d'un cube, d'une surface plane, d'une ligne puis d'un cercle. Nous voyons l'apprentissage à son début, les poids des neurones ayant été définis aléatoirement, puis la topologie en cours et en fin d'apprentissage.*

Apprentissage Lors de la phase d'apprentissage, un réseau neural gas effectue la représentation de l'espace des entrées dans un espace de dimension plus réduite, selon une méthode proche de celle décrite pour Kohonen. Pour chaque exemple présenté en entrée, un neurone est déclaré vainqueur et les neurones les *plus proches* du vecteur d'entrée voient leurs poids modifiés afin de se rapprocher de l'exemple présenté.

3. Learning Vector Quantization

L'apprentissage de la topologie se fait exclusivement au niveau du neurone vainqueur. Si elle n'existe pas, une connexion est créée entre le neurone vainqueur et son second, l'âge de cette connexion est mis à zéro. Les âges de toutes les connexions du neurone vainqueur sont incrémentés, puis les connexions trop *vieilles*, celles dont l'âge dépasse un seuil fixé, sont détruites, car jugées non représentatives de l'espace des entrées.

Cette méthode de construction des connexions permet de ne retenir dans la topologie que les connexions fréquemment validées par l'ensemble d'apprentissage. Cette méthode permet une meilleure dispersion des neurones dans l'espace d'entrée, et par là même, une représentation plus précise de celui-ci. Le réseau peut en effet prendre une topologie de grille en certains endroits de l'espace, mais être seulement en ligne en d'autres et même laisser certains neurones *isolés* représenter seuls une partie des données si cela est nécessaire, ce qui est impossible à la topologie imposée des cartes de Kohonen (voir figure 2.13).

Le modèle Growing neural gas Il existe un modèle dérivé de neural gas, qui en est la version incrémentale : *Growing neural gas* [Fritzke, 1995]. Ce modèle apprend donc les poids de ses neurones et la topologie du réseau est construite suivant la méthode de *neural gas*. En revanche la taille du réseau, le nombre de neurones de celui-ci, n'est pas fixée au départ de l'apprentissage.
Avec cet algorithme, l'apprentissage commence avec un réseau ne comprenant que deux neurones puis des neurones sont ajoutés au réseau près des neurones commettant le plus d'erreurs cumulées.

Avec cet algorithme tout est construit à l'apprentissage : taille du réseau, répartition des neurones et topologie du réseau. Le comportement et les propriétés de *Growing neural gas* sont identiques à *neural gas*

2.3 Les modèles d'inspiration biologique

Comme nous l'avons vu avec les modèles de cartes auto-organisatrices, certains aspects des recherches en neurobiologie inspirent le monde et les architectures connexionnistes. A la suite du neurone formel, à l'origine des réseaux de neurones, qui est directement inspiré du neurone biologique, de nombreux chercheurs tentent d'extraire de nouveaux modèles des travaux menés sur le cerveau par les neurobiologistes, mais aussi des psychologues, des biochimistes ou des physiologistes. Ces recherches ont essentiellement pour objectifs de valider les modèles "observés", d'étendre les propriétés des réseaux de neurones à l'aide de nouvelles architectures et stratégies d'apprentissage, d'approcher et de comprendre les traitements effectués par les différentes structures nerveuses du cerveau.

En règle générale, les architectures issues de la biologie représentent des modèles à plusieurs granularités, chacune de ces granularités s'appuyant sur des caractéristiques topologiques particulières.

Par exemple, le modèle de cortex associatif de Hervé Frezza-Buet [Frezza-Buet, 1999] fait suite à de précédents travaux inspirés des données biologiques [Alexandre et Guyot, 1995; Durand, 1995].

L'unité de base n'est plus le neurone, formel ou continu, mais la *maxicolonne*, un ensemble de *colonnes corticales* (voir section 1.2.6). Suivant la modalité de l'information qu'elles traitent, les unités sont regroupées en cartes au sein desquelles elles sont totalement interconnectées. La capacité de ces modèles à combiner des informations de différentes modalités repose sur l'interconnexion des cartes. Dans l'ensemble, une unité possède par conséquent un nombre important

de connexions (voir figure 2.14). De plus, pour se spécialiser, une carte crée dynamiquement de nouvelles unités.

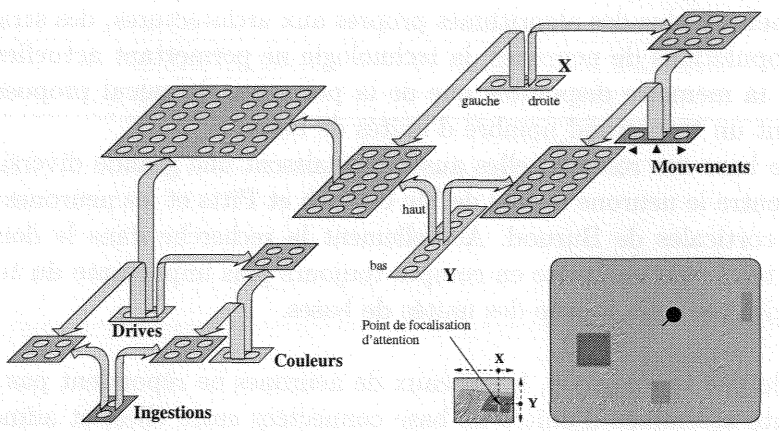


FIG. 2.14 – *Un exemple de modéle biologique. L'architecture associative de contrôle d'un robot de Hervé Frezza-Buet.*

Ces modèles sont appliqués au traitement perceptivo-moteur de robots afin de les doter de capacités comportementales élaborées. Ils intègrent des traitements statistiques pour l'analyse des perceptions mais aussi la composante motivationnelle qui dirige le comportement vers un but.

La complexité et le nombre des unités de ces modèles, ajoutés au grand nombre de connexions, rendent ces modèles difficiles à tester en raison des temps de calcul engendrés.

2.4 Conclusion : Vers une définition structurelle des réseaux connexionnistes

Nous avons effectué un rapide survol du domaine des réseaux de neurones. Notre propos étant plus structurel que qualitatif, nous ne cherchons pas à discuter ou à évaluer les différents types de réseaux de neurones dans leurs champs d'application. Nous souhaitons plutôt définir les réseaux du point de vue du bâtisseur. Comment se développe informatiquement un réseau ? Quelles sont les caractéristiques essentielles des différentes unités qui le composent ? Voici les questions auxquelles nous nous attachons. Ainsi nous ne discuterons pas des mérites comparés des différentes architectures ou algorithmes d'apprentissage mais nous nous intéresserons à leurs particularités topologiques.

Bien qu'issus de l'étude des paradigmes biologiques du cerveau, les modèles des réseaux de neurones artificiels sont essentiellement des modèles statistiques. Ces architectures ont plus comme objectif de résoudre des problèmes mathématiques, comme l'approximation de fonctions ou la classification, que de simuler des propriétés cérébrales, comme les capacités de mémoires ou d'abstraction.

Il est important de noter qu'il n'existe pas d'architecture connexionniste idéale, l'utilisateur doit choisir son architecture en fonction du problème à résoudre, en tenant compte des qualités, mais aussi des défauts de chacune d'entre elles. Ainsi les perceptrons multi-couches sont difficiles à paramétrer, les algorithmes incrémentaux induisent un risque d'explosion du nombre de neurones

créés et un risque de sur-apprentissage (apprentissage *par cœur*), l'élagage nécessite un grand nombre de paramètres et d'itérations du corpus d'apprentissage. De même les modèles biologiques doivent développer, en plus des algorithmes propres aux architectures, des stratégies permettant de simuler les populations de neurones, la technologie ne permettant actuellement pas, tant du point de vue de la mémoire disponible que de la puissance de calcul proposée, de simuler des réseaux contenant un trop grand nombre d'unités de bases.

Les unités de base des réseaux, elles aussi, connaissent une grande diversité. Il ne reste que peu d'analogies entre le neurone formel de Mc Culloch et Pitts et les neurones à fuites de Taylor ou les colonnes corticales de Burnod. Actuellement la recherche dans le domaine des réseaux connexionnistes tend vers une prise en compte toujours plus importante du temps, ce qui passe par une complexité toujours accrue des unités de bases.

D'un point de vue topologique, les réseaux de neurones ne répondent pas à des critères formels précis. Toute assemblée d'unités de base connectées entre elles, et admettant des entrées et des sorties, peut être définie comme entité connexionniste, pour peu que ses unités de base agissent de manière autonome, c'est à dire qu'elles ne définissent leur sortie qu'en fonction de leurs entrées et de critères propres locaux, des variables locales par exemple. Ainsi nous n'avons présenté ici que les modèles classiques, nous permettant de définir un vocabulaire sur le domaine et les particularités architecturales, à partir desquelles toute configuration devient possible.

Nous pouvons en fait tirer trois conclusions de cette partie du manuscrit.
Les réseaux de neurones artificiels sont coûteux en temps de calcul, essentiellement lors de leur phase d'apprentissage pour les modèles statistiques et dans les besoins nécessaires à la modélisation pour les modèles biologiques.
Les réseaux de neurones artificiels fonctionnent sur la base de calculs distribués. Ils sont constitués d'une somme, pouvant être élevée, d'unités de base effectuant des opérations localement sur leurs entrées avant de les transmettre au monde extérieur. Ces opérations locales, si elles sont le plus souvent des somme pondérées, peuvent être beaucoup plus variées et complexes. De plus les neurones peuvent distinguer les diffèrentes entrées qu'ils reçoivent, en fonction du type de la connexion concernée, et ainsi adopter un comportement différent selon les entrées.
En terme de topologie, s'il existe de grandes familles, réseaux à couches et réseaux dynamiques par exemple, toutes les combinaisons entre les entités de base sont possibles. La seule constante est qu'une topologie de réseau connexionniste est créée par les connexions des neurones qui le composent et qu'une connexion est un lien entre la sortie d'un premier neurone et une entrée d'un second, lien permettant au second un accès à la valeur de la sortie du premier.

Pour aider les utilisateurs et concepteurs de réseaux connexionnistes, il serait donc souhaitable de travailler sur ces trois points. Leur faciliter la programmation en mettant à leur disposition un outil permettant de construire aisément leurs modèles, en ne se préoccupant que des aspects spécifiquement connexionnistes de leurs modèles, et leur permettre d'accélérer leurs applications afin de pouvoir développer des réseaux de taille plus conséquente tout en gardant des temps d'exécution acceptables.

Le développement d'un simulateur dédié au connexionnisme peut être une réponse aux besoins en développement tandis que l'utilisation de machines parallèles généralistes semble pouvoir répondre aux problèmes de temps et de taille des réseaux de neurones artificiels.

Chapitre 3

Les spécificités de la programmation parallèle

Le monde des machines parallèles, s'il est, comme nous le verrons dans ce chapitre, vaste et multiple, est régi par un vocabulaire commun, vocabulaire rendant compte de propriétés et difficultés communes aux différentes architectures.

Avant d'aborder, plus ou moins succinctement, les principales architectures parallèles, nous présentons les spécificités de la programmation parallèle. Nous introduirons donc les notions de granularité des applications et des machines parallèles, les modes de communication entre les différents processeurs de la machine, la synchronisation de ces processeurs et l'équilibrage de charge.

Notre propos étant la recherche d'un outil portable, nous ne parlerons ici que de machines "généralistes", et donc pas des architectures spécifiques et de leurs particularités.

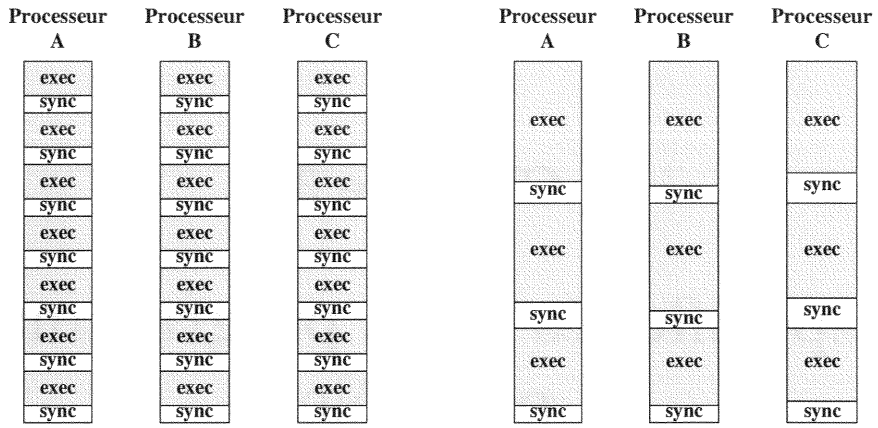
Lors de l'exécution d'une application séquentielle, la totalité des calculs affectés à cette application sont effectués par un **unique processeur**. Charge est au programmeur d'adapter son implantation à cette caractéristique, qui lui est devenue naturelle.

Lors d'une exécution sur machine parallèle, généralement effectuée pour abaisser le temps d'exécution de l'application, **plusieurs processeurs** sont utilisés. Les calculs sont effectués par ces différents processeurs et le programmeur doit répartir les calculs sur les différents processeurs. Cet apport de puissance de calcul entraîne donc des modifications dans la conception de la programmation des applications.

3.1 Parallélisme et granularité

La granularité d'une application fait référence à la division de la tâche à accomplir en calculs parallèles autonomes. La granularité donne une information sur la quantité de calculs effectuée par les processeurs entre deux opérations de communication ou de synchronisation.

Un parallélisme à grain fin désigne une application à bas niveau de parallélisme, constituée d'une somme de tâches parallèles relativement courtes. Les applications parallèles à grain fin désignent donc des applications effectuant fréquemment des opérations de synchronisation ou de communication entre les différents processeurs (voir figure 3.1(a)).



(a) Application à grain fin (b) Application à gros grain

FIG. 3.1 – *Granularité. Dans les applications à grain fin les périodes d'exécution autonome (*exec*) sont courtes avec de fréquentes opérations de synchronisation (*sync*). Les périodes d'exécution autonome sont plus longues dans les applications à gros grain.*

Un parallélisme à gros grain désigne, à l'opposé, une application à haut niveau de parallélisme, avec de grandes périodes parallèles autonomes, donc avec une fréquence de communication et synchronisation assez faible (voir figure 3.1(b)) comparée aux périodes de calcul des processeurs.

Une programmation à grain fin d'un réseau de neurones consisterait, par exemple, à placer un neurone par processeur, un parallélisme à gros grain consisterait à recopier le réseau sur chacun des processeurs disponibles et à distribuer les exemples sur ces processeurs. Dans le premier cas, chaque communication entre les neurones entraîne une opération de communication entre deux processeurs. Dans le second cas les communications entre les processeurs n'auront lieu que pour comparer/ajuster les résultats, donc beaucoup plus rarement. Entre ces deux extrêmes théoriques, toute granularité intermédiaire est possible.

3.2 Loi de Amdhal

Les raisons essentielles de l'utilisation d'une machine parallèle, donc de la parallélisation d'un programme, sont d'ordre temporel. Une implantation sur machine parallèle doit accélérer l'exécution d'un programme. La parallélisation peut rendre réalisables certaines applications, en réduisant le temps d'exécution de plusieurs semaines à quelques jours, pour des calculs lourds (bioinformatique, météo, physique, chimie et mathématiques par exemple) [Boniface, 1994].

Une application peut aussi être parallélisée pour approcher le "temps réel", il s'agit ici d'applications rapides qu'il faut rendre "instantanées", c'est plus souvent le cas de systèmes embarqués. Par exemple en réduisant suffisamment les temps de planification de son environnement, un robot pourra circuler tout en effectuant ses calculs [Laroche *et al.*, 2000].

Dans tous les cas, l'idéal recherché lors de la parallélisation d'une application est de diviser le temps de son application par le nombre de processeurs utilisés (voir équation 3.1) .

$$\text{Temps d'Exécution Parallèle} = \frac{\text{Temps d'Exécution Séquentielle}}{\text{Nombre de Processeurs Utilisés}} \tag{3.1}$$

L'accélération idéale d'une parallélisation serait donc proche du nombre de processeurs utilisés.

$$\begin{aligned} \text{Accélération Idéale} &= \frac{\text{Temps d'Exécution Séquentielle}}{\text{Temps d'Exécution Parallèle}} \\ &= \text{Nombre de Processeurs Utilisés} \end{aligned} \tag{3.2}$$

Mais ces équations sous-entendent que la totalité des opérations effectuées par l'application s'exécutent en parallèle. Amdhal [Amdhal, 1967] remarque qu'une application possède toujours une partie non parallélisable, sa *fraction séquentielle*, ce qui implique que l'accélération d'une application parallèle, exécutée sur P processeurs, est plafonnée à une accélération maximale, accélération décrite dans l'équation 3.3.

$$\text{Accélération Maximale} = \frac{\text{Temps Séquentiel}}{\frac{\text{Part Parallèle}}{P} + \text{Part séquentielle}} \tag{3.3}$$

La "loi de Amdhal" indique donc qu'il existe une limite aux performances de la parallélisation d'un programme. Soit f_s la proportion non parallélisable d'une application. Si $S(P)$ représente l'accélération de l'application sur P processeurs, les performances de la parallélisation de l'application seront limitées, quel que soit le nombre de processeurs utilisés, comme décrit dans l'équation 3.4.

$$\begin{aligned} S(P) &= \frac{1}{f_s + \frac{(1-f_s)}{P}} \\ &\leq \frac{1}{f_s} \end{aligned} \tag{3.4}$$

A titre d'exemple, un programme comprenant **une fraction séquentielle de 10%**, ne pourrait espérer mieux, d'après l'équation 3.4, qu'une accélération de **5,26** sur **10 processeurs**, et quel que soit le nombre de processeurs utilisés l'accélération sera toujours inférieure à **10**.

3.3 La communication entre processeurs

Lors de l'exécution d'une application parallèle, quel que soit le type de machine parallèle utilisée, chaque processeur manipule des données. L'une des spécificités de la programmation parallèle est la transmission des données d'un processeur à un autre. Il s'agit de se demander comment une donnée utilisée par un certain processeur peut être manipulée, en lecture et/ou écriture, par un ou plusieurs autres processeurs de la machine parallèle.
Il existe en fait deux paradigmes principaux permettant de communiquer des données d'un processeur à un (aux) autre(s) sur les machines parallèles : *l'envoi de messages* ou *le partage de mémoire*.

3.3.1 La communication par envoi de messages

La communication par envoi de messages considère que chaque processeur est propriétaire des données qu'il manipule. Dans ce protocole les communications sont explicites. Si l'exécution d'un processeur nécessite l'utilisation d'une variable gérée par un processeur différent, la donnée doit être envoyée par le processeur hôte de la variable au processeur qui la demande, ce dernier devant, quant à lui, la recevoir explicitement (voir figure 3.2). De même un processeur peut envoyer une, ou plusieurs, de ses données à plusieurs processeurs par un même appel, mais chacun d'entre-eux doit demander à la recevoir. Il faut noter que l'ordre chronologique des demandes d'envoi et de réception d'un message n'a aucune importance.

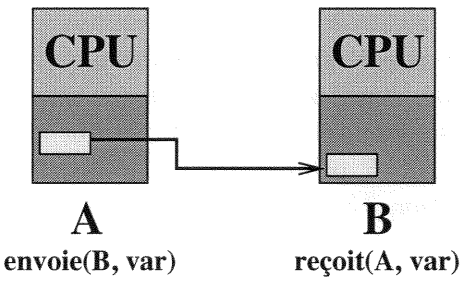


FIG. 3.2 – *L'obtention de la donnée* var *par le processeur* **B** *nécessite deux opérations explicites. 1) Le processeur* **A** *demande explicitement l'envoi d'un message contenant* var *au processeur* **B**. *2) Le processeur* **B** *demande explicitement la réception d'un message contenant* var *du processeur* **A**.

Du point de vue logiciel, deux grands standards de communication par envoi de messages existent : PVM (Parallel Virtual Machine[4]) [Geist, 1994] et MPI (Message Passing Interface[5]) [Gropp *et al.*, 1994; Gropp, 1999]. Ces deux bibliothèques ont l'avantage de rendre les programmes les utilisant portables sur les différentes architectures parallèles. Ces deux paradigmes, très répandus dans la "communauté parallèle", ont donné lieu à diverses sur-couches, des interfaces facilitant ou spécialisant l'écriture des envois de messages (voir, par exemple, PARA ++ [Coulaud et Dillon, 1996]).

Un grand avantage de l'implantation d'un programme par envois de messages effectué avec des interfaces portables comme MPI et PVM est d'offrir à l'utilisateur la possibilité d'exécuter son programme sur différentes architectures de machines parallèles, mais aussi sur réseau de stations séquentielles classiques, réseau pouvant même contenir des machines d'architectures différentes.

L'utilisation de ce paradigme entraîne des difficultés de programmation, le programmeur devant pouvoir localiser ses variables à tout moment de l'exécution d'une application. Contrairement à la programmation séquentielle classique, une donnée ne dispose plus d'un adressage fixe, elle dispose d'une localisation supplémentaire : son processeur hôte. La programmation utilisant ce paradigme demande donc, de la part du programmeur l'effort de bien répartir ses données sur les mémoires des différents processeurs. Il se doit aussi de bien définir les besoins de communication de données entre les différents processeurs utilisés, afin que chaque processeur nécessitant une donnée ne lui appartenant pas en propre puisse la recevoir explicitement de son processeur hôte.

Par là même, la programmation parallèle utilisant la communication par envoi de messages entre

4. Machine Virtuelle Parallèle
5. Interface d'Envois de Messages

les différents processeurs nécessite de profondes modifications des algorithmes séquentiels de la part des programmeurs.

Les programmes développés en communication par envoi de messages ont néanmoins l'avantage d'être *scalables* [6], ce qui signifie que les performances d'une application, en terme d'accélération, sont conservées à l'ajout de nouveaux processeurs [7]. Ce type de programmation parallèle est aussi le plus utilisé par la communauté parallèle. Il bénéficie essentiellement de l'ancienneté de ses standards en comparaison de ceux de la programmation par partage de mémoire et de sa portabilité. Cette portabilité inclut les réseaux de stations. Ainsi des standards comme PVM et MPI permettent aux utilisateurs l'accès au parallélisme sans besoin d'investir dans de coûteuses machines parallèles.

3.3.2 La communication par partage de mémoire

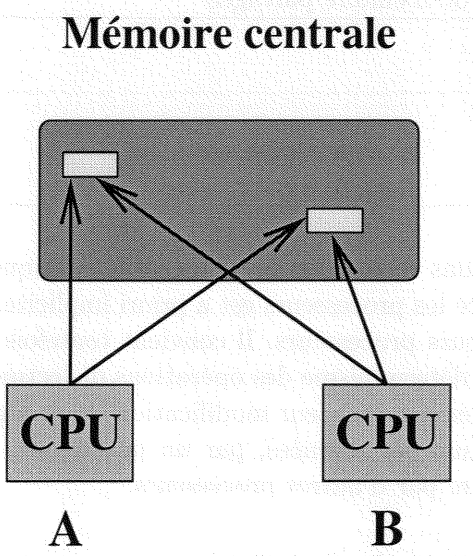


FIG. 3.3 – *Communication par partage de mémoire. L'architecture quelle qu'elle soit, est vue, pour le programmeur, comme une somme de processeurs, disposant d'un même espace mémoire.*

Le second paradigme de communication entre processeurs d'une machine parallèle est la communication par partage de mémoire. Une application à communication par partage de mémoire considère la machine parallèle comme une somme de processeurs ayant accès à une mémoire principale unique (voir figure 3.3).

Avec ce paradigme de communication, comme pour la programmation séquentielle, chaque variable dispose d'un adressage unique et est accessible en lecture comme en écriture par tous les processeurs de la machine. Par cette caractéristique, et en comparaison de la communication par envoi de messages, cette méthode de communication est souvent considérée comme la plus simple à utiliser, car la plus proche de la programmation séquentielle classique. Un processeur n'ayant plus besoin de s'attribuer explicitement une variable pour l'utiliser, ce qui résout le problème de placement des données inhérent à l'envoi de messages, la programmation n'impose théoriquement

6. extensibles. Le terme anglais, mot clé du parallélisme, sera utilisé dans ce manuscrit.

7. Par exemple, si un programme est exécuté deux fois plus rapidement sur deux processeurs que sur un processeur unique, il est possible d'espérer une accélération d'exécution proche de quatre sur quatre processeurs, de huit sur huit processeurs, etc.

que peu d'adaptation des algorithmes au parallélisme.

La parallélisation à l'aide de ce type de communication entre les processeurs comporte néanmoins des difficultés, toujours liées à la gestion des données.
L'algorithme 3.1 présente un exemple de problème pouvant se poser en programmation par mémoire partagée. Lorsque l'algorithme 3.1 est exécuté par un processeur A, rien ne garantit la valeur de *variable* rendue par A, alors que l'on attend *variable* valant *667*. En effet, *variable* étant une variable partagée, un autre processeur a pu l'utiliser et changer sa valeur entre l'exécution de la condition et celle de l'instruction par le processeur A. De plus, si *variable* est partagée et qu'un processeur concurrent l'utilise en lecture, il est clair qu'il n'obtiendra pas la même donnée s'il effectue son opération de lecture avant ou après l'instruction. De même que se passe-t-il si un processeur demande à lire *variable* pendant l'incrémentation?

Algorithme 3.1 Problème de mémoire partagée

```
si (variable = 666)                                   Condition
 alors
   variable = variable + 1                            Instruction
fin si
```

La gestion des communications reste donc toujours problématique. Avec ce paradigme, la communication des données entre les processeurs est a priori implicite, les données "à communiquer" sont ici partagées par plusieurs processeurs. Il convient toutefois de s'assurer du déterminisme des applications, et donc du déterminisme des opérations de lecture/écriture des données en protégeant les accès aux variables pendant leur modification (voir algorithme 3.2), ou en s'assurant que les phases de modification des données, par un processeur, d'une application seront bien séparées des phases de lecture par d'autres processeurs.

L'algorithme 3.2 est un exemple de méthode permettant de rendre déterministe l'exemple de l'algorithme 3.1. Pour éviter toute ambiguïté, *variable* est interdite en lecture comme en écriture, pour tous les autres processeurs du système, pendant l'exécution de l'algorithme par le processeur A, par la fonction *Protège()*. Elle est rendue de nouveau accessible par l'appel à la fonction *Libère()*. Entre ces deux appels, *variable* est propriété exclusive du processeur effectuant cette exécution, cette phase est appelée *section critique*.

Algorithme 3.2 Exemple de protection d'une variable en mémoire partagée

```
Protège(variable)
si (variable = 666)                                   Condition
 alors
   variable = variable + 1                            Instruction
fin si
Libère(variable)
```

Comme pour la programmation par envoi de messages, ce protocole de communication est indépendant du support architectural. Il existe en effet des interfaces logicielles de simulation de mémoire partagée sur machines parallèles à mémoire distribuée (voir section 4.2.1). Dans ce cas,

la communication par partage de mémoire sera généralement la sur-couche d'une implantation à envoi de messages [Kermarrec et Morin, 1997; Mentré et Priol, 1998; Li, 1986a]. Les performances obtenues seront souvent beaucoup moins bonnes que sur des architectures à mémoire partagées.

Deux standards semblent s'imposer dans le domaine de la programmation parallèle en communication par partage de mémoire, mais sur machines parallèles à mémoire partagée uniquement : Les *Threads Posix*[Nichols *et al.*, 1996], norme développée pour les machines séquentielles et stations multiprocesseurs qui commence à être portée sur architectures parallèles, et surtout *OpenMp* [Dagum et Menon, 1998] développé par les grands constructeurs de machines parallèles et devant devenir la norme pour la programmation sur machines parallèles à mémoire partagée. Cette dernière est une librairie disponible sur les langages C et Fortran offrant des outils permettant de générer des *threads*[8] constructeurs, donc parfaitement adaptés à la machine cible.

3.4 La synchronisation entre les processus

La programmation parallèle entraîne le besoin de synchronisation entre les processeurs. Il est en effet souvent utile pour un processeur de s'assurer de la disponibilité de certains résultats, calculés par d'autres processeurs, pour finir son exécution. Pour ce faire, il est nécessaire de *synchroniser* les processeurs, i.e. d'attendre que certains processeurs, explicitement désignés par le programmeur, soient tous arrivés à un point de rendez-vous pour reprendre le cours de l'exécution.

Par exemple, si l'on cherche à calculer une distance minimale entre différents points, une solution possible est que chaque processeur calcule une distance entre deux points et l'un des processeurs, le processeur A, détermine la distance minimale obtenue. Pour ce faire, le processeur A doit s'assurer de la fin du calcul de tous les autres processeurs avant de lancer son propre calcul, ce qui nécessite une synchronisation (voir algorithme 3.3).

Algorithme 3.3 Exemple de synchronisation

```
pour Tous les processeurs faire
    détermination de la distance
    Synchronisation                          Attente de tous les processeurs
fin pour
pour Le processeur A faire
    Détermination de la distance minimale
fin pour
```

Il existe diverses formes de synchronisation dans un programme. Un processeur peut juste signaler qu'il est passé par un point défini de son algorithme, et continuer son exécution. Un processeur peut être obligé d'attendre l'arrivée d'autres processeurs à un certain point pour continuer son exécution, on parle dans ce cas de *barrière de synchronisation.* Il existe d'autres cas comme, par exemple, les synchronisations dues à la protection d'une variable (voir algorithme 3.2), quand un processeur attend la fin de la modification d'une variable pour pouvoir l'utiliser à son tour.

Si la synchronisation est un élément incontournable de la programmation parallèle, elle peut poser de nombreux problèmes d'efficacité, une synchronisation entraînant une suspension d'exé-

8. Processus légers

cution pour le processeur. Par exemple, lorsqu'un processeur arrive à une barrière de synchronisation, il reste inactif jusqu'à sa libération, libération consécutive à l'arrivée de tous les processeurs concernés par cette barrière. Lors de l'implantation de synchronisations, le programmeur doit donc équilibrer au mieux la charge des processeurs entre deux synchronisations pour limiter la perte en efficacité (voir section 3.5).

La manipulation de la synchronisation est différente selon l'architecture et le mode de communication choisis.
Du point de vue de la programmation, c'est avec la communication par envoi de messages que la synchronisation est la plus souple. Elle peut être implicitement intégrée aux communications entre les processeurs avec, par exemple, des réceptions bloquantes (le processeur cesse toute activité à la demande de réception d'un message tant que ce message ne lui est pas parvenu). La résolution des problèmes de communication des données entraîne souvent celle des problèmes de synchronisation. En communication par partage de mémoire, les méthodes de synchronisation sont explicites et moins nombreuses. Elles se résument le plus souvent à des barrières de synchronisation et des protections de variables par exclusion mutuelle.
Du point de vue technologique, une synchronisation de deux processeurs sur machine à mémoire partagée est moins coûteuse en temps. Elle ne nécessite que la modification d'une variable, au lieu de plusieurs échanges de messages pour les machines à mémoire distribuée [Papadopoulo, 1997].

3.5 L'équilibrage de charge

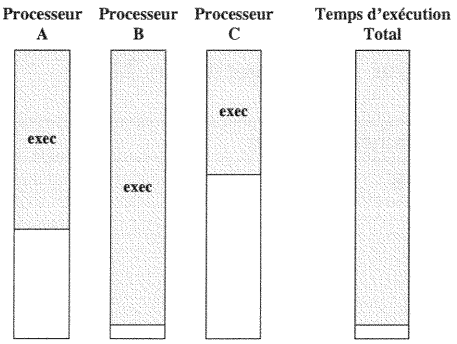
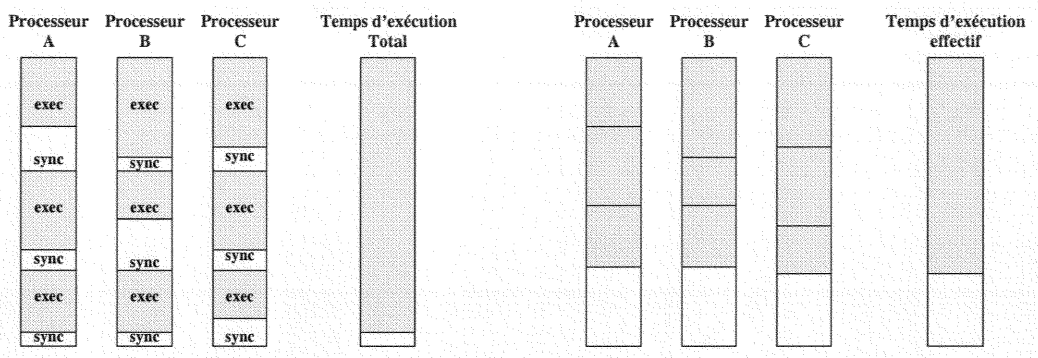


FIG. 3.4 – *Exemple de défaut d'équilibrage simple. Les parts de calculs à effectuer sont mal réparties entre les différents processeurs, les processeurs A et C ayant beaucoup moins de calculs à effectuer que le processeur B. Le temps d'exécution total de l'application étant le temps d'exécution du processeur ayant la plus grosse charge de calcul, une meilleure répartition des calculs sur les processeurs dimunuerait le temps d'exécution total.*

Dans la recherche de performances inhérente à l'utilisation d'une machine parallèle, l'équilibrage de charge est un facteur essentiel. Il représente la répartition des calculs sur les processeurs utilisés. Nous avons vu avec Amdhal (voir section 3.2) que la recherche de performance, de division du temps d'exécution par répartition des calculs, connaissait une limite formelle. Pour approcher le temps d'exécution minimal possible, il est nécessaire de répartir au mieux les calculs d'une application sur les différents processeurs utilisés pour son exécution. Le problème de l'équilibrage de charge réside donc dans cette répartition. Il consiste à s'assurer du fait qu'un ou plusieurs processeurs ne seront pas en attente d'un ou plusieurs autres processeurs durant

l'exécution, l'attente pouvant être la conséquence d'une synchronisation (voir figure 3.5) ou de la mauvaise répartition de la charge sur les processeurs (voir figure 3.4).



(a) Description de l'exécution

(b) Temps effectifs de calcul

FIG. 3.5 – *Exemple de mauvais équilibrage de charges. La charge est répartie de manière équitable sur les processeurs, mais les calculs entre chacune des barrières de synchronisations ne sont pas équilibrés. La conséquence en est un temps d'exécution très supérieur aux temps de calcul effectifs.*

3.6 Conclusion

Nous avons évoqué dans ce chapitre les aspects spécifiques de la programmation parallèle. Ces notions, si elles éloignent la programmation parallèle de la programmation séquentielle classique, doivent être prises en compte pour espérer obtenir de bonnes performances sur machines parallèles.

Au delà de ce vocabulaire, il n'existe que peu de généralités sur les implantations et les méthodes de programmation parallèles efficaces. L'obtention de performances est en effet extrêmement dépendante de l'application concernée et de l'architecture de la machine cible.

Chapitre 4

Les architectures parallèles

Le parc des différentes machines parallèles *généralistes* se divise en deux classes d'architecture, les architectures *SIMD* et *MIMD*. Ces architectures se distinguent autant par leurs caractéristiques techniques que par les techniques utilisées pour leur programmation et leurs domaines de compétences respectifs. De ces deux types d'architectures, l'architecture MIMD est la plus riche en variété. Elle compte en effet, sous la même appellation générique, de nombreuses architectures différentes, essentiellement différenciées par la construction de leur zones mémoires.

4.1 Les architectures SIMD

Nous passerons rapidement sur ce type d'architecture parallèle, le type d'ordinateur qui le supporte semblant être en passe de disparaître dans le monde des machines parallèles. Nous ne présenterons donc que les généralités techniques de ce modèle puis nous présenterons pour l'exemple une machine, la MasPar [Blank, 1990], et la méthode de programmation sur celle-ci. Le lecteur pourra obtenir de plus amples renseignements sur cette architecture d'ordinateur dans [Hillis, 1985], qui présente la *Connection Machine* l'une des machines SIMD, ayant été la plus répandue, avec la MasPar.

4.1.1 Spécificités techniques

Les machines SIMD sont des machines parallèles à grain fin. Au niveau architectural, ces machines sont composées de très nombreux processeurs, plusieurs centaines, voire plusieurs milliers (la MasPar pouvait en contenir jusqu'à 4096). La particularité de ces processeurs est d'être simples : ils n'ont pas une très forte puissance de calcul. De plus, chacun de ces processeurs dispose d'un petit espace mémoire propre. La puissance de ce type d'architecture parallèle repose sur le nombre de processeurs disponibles, sur la topologie du réseau de processeurs, sur la vitesse de leur communication et non pas sur la puissance de chacun des processeurs, au contraire des architectures de type MIMD. Les processeurs de la machine parallèle, appelés *Processeurs Elémentaires* sont dénués de *contrôleur d'instruction*, cette tâche étant attribuée à un processeur spécifique, appelé *processeur de contrôle*. Communément, ce processeur gère les tâches effectuées par les différents processeurs élémentaires et les synchronisations entre ceux-ci.

Techniquement SIMD signifie *Simple Instruction Multiple Data* soit instruction unique sur données multiples. Avec ce type d'ordinateur parallèle, le programmeur dispose de variables locales aux processeurs et de variables globales. Les processeurs disposent d'un accès, en lecture, aux variables locales des autres processeurs du système, variables qui seront acheminées à travers

les liens de communication de la machine, suivant la topologie de celle-ci.

L'utilisation classique de ce type d'ordinateur est donc d'envoyer aux processeurs élémentaires, par l'intermédiaire du processeur de contrôle, un ensemble de tâches à effectuer. Les processeurs élémentaires effectueront donc simultanément la même tâche, mais chacun sur ses donnés propres, puis seront synchronisés avant d'effectuer la tâche suivante (voir algorithme 4.1).

Algorithme 4.1 Modèle général de programmation SIMD

Faire en parallèle
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Fin parallèle

Ce synchronisme tacite des processeurs, dit 'pas à pas', géré directement par le système est une des particularités de la programmation sur machine parallèle de type SIMD. Le fonctionnement des processeurs est strictement synchrone. A l'exécution de l'algorithme 4.1, les processeurs élémentaires exécuteront simultanément *instruction 1*, seront synchronisés par le processeur de contrôle, exécuteront simultanément *instruction 2*, etc.
Le synchronisme 'pas à pas' et l'exécution simultanée d'une même instruction par les différents processeurs dispense le programmeur de la gestion de l'équilibrage de charge et du synchronisme.

La section 4.1.2 présente un exemple de l'utilisation de ce type de machine parallèle : La MasPar.

4.1.2 Un exemple de machine SIMD : La MasPar

Spécificités techniques

La figure 4.1 présente une vision schématique de l'architecture de la MasPar. La machine se présente comme une grille de processeurs élémentaires, en fait un tore, contrôlés par un processeur principal et ayant accès à une mémoire principale collective. Les processeurs élémentaires sont des processeurs 4 bits et ils disposent chacun de 64 kilo-octets de mémoire propre.

Chacun des processeurs élémentaires est directement connecté à ses huit voisins directs et dispose, par ces connexions, d'un accès en lecture aux variables locales de tous les autres processeurs de la grille. La gestion de ces connexions se fait à partir de fonctions basés sur des mots clés comme *Nord*, *Sud*, *Est*, *Ouest*, *Nord-Est*, etc pour indiquer les directions des connexions.

Techniques de programmation

Comme il a été vu dans la section 4.1.1, la programmation sur ce type d'architecture se fait en demandant l'exécution en parallèle d'un certain nombre de tâches.

Dans l'algorithme 4.2, le but est de placer, pour chaque processeur élémentaire, dans sa variable locale *LToto*, le maximum du *LToto* local, du *LToto* de son voisin de droite et d'un seuil *Seuil*. Cette dernière variable est une variable globale du programme, c'est à dire stockée dans la mémoire principale de l'ordinateur et accessible par tous les processeurs élémentaires.

L'instruction :

Plurale *LToto*

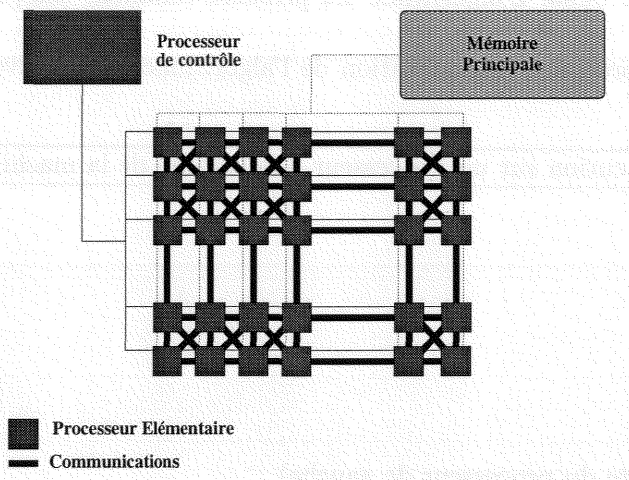


FIG. 4.1 – *Exemple d'architecture SIMD, la MasPar.*

Algorithme 4.2 Exemple d'algorithme sur machine SIMD
LToto = MAX(*LToto*, *LToto[processeur de droite]*, *Seuil*)

But : Calcul, pour chaque processeur, du maximum entre sa variable *LToto*, celle de son voisin de droite et *Seuil*

plurale LToto
plurale LTmp
globale Seuil

Faire en parallèle
LTmp = MAX(Est[1].LToto, Seuil) *Instruction 1*
LToto = MAX(LToto, LTmp) *Instruction 2*
Fin parallèle

signifie que chaque processeur disposera d'une variable locale *LToto*. Cette variable pourra donc avoir des valeurs différentes sur différents processeurs.

Chaque processeur élémentaire dispose donc de ses variables locales *LToto* et *LTmp* et peut lire la variable globale *Seuil*. L'algorithme 4.2 présente comment chaque processeur met à jour sa variable locale.

L'algorithme 4.3 propose l'interprétation de l'algorithme 4.2 au niveau d'un processeur élémentaire du système.

Algorithme 4.3 Exécution sur un Processeur Elémentaire de la machine

locale LToto
locale LTmp
locale tmp
globale Seuil

Début
  tmp = **Lire**(*LToto* du processeur de gauche) *Est[1].LToto*
  LTmp = Max(tmp, Seuil) *Instruction 1*
  Synchronisation
  LToto = Max(LToto, LTmp) *Instruction 2*
fin

Chacun des processeurs élémentaires effectuera l'instruction 1. Il lira donc la variable *LToto* du processeur situé à sa droite, il définira le maximum entre cette valeur et le seuil *Seuil*. Il attendra ensuite la fin de cette exécution pour tous les processeurs avant d'effectuer l'instruction 2.

Il faut noter la présence de la variable *LTmp* qui permet la cohérence de la lecture de *LToto*. En effet, si l'écriture se faisait directement dans la variable *LToto*, rien ne permettrait de garantir que la variable *LToto* lue par le processeur "à droite" n'a pas déjà été modifiée par son processeur hôte.

4.1.3 Conclusion sur les machines SIMD

Les machines parallèles de type SIMD sont performantes pour des applications à grain fin, sur les problèmes à structures régulières où la même instruction s'applique à des sous-ensembles de données aisément parallélisables. Elles sont très adaptées aux problèmes de parallélisme de données, c'est à dire à la répartition des données à traiter sur une somme de processeurs effectuant exactement le même traitement (calculs matriciels élémentaires ou résolutions de systèmes linéaires par exemples). Elles offrent aux programmeurs, pour ce type d'applications, une sémantique de programmation les exonérant des difficultés de gestion de la cohérence de données, de l'équilibrage de charge (tous les processeurs effectuent la même tâche simultanément) et des synchronisations. Sur des problèmes conditionnels, ces architectures sont beaucoup moins performantes.

Ces architectures nécessitent néanmoins un mode de programmation très différent de la programmation séquentielle, et fortement dépendant de la topologie de la machine. Ces machines souffrent d'un manque de stabilité des architectures et d'un manque de langages performants. De plus, l'évolution des processeurs séquentiels classiques du marché, du type Pentium, ne peut

être suivie par les constructeurs, le coût de recherche et de développement étant très élevé pour des processeurs spécifiques [Parhami, 1995].

Ces difficultés rendent les applications développées sur machines SIMD coûteuses en temps de programmation et peu portables, avec des gains de temps obtenus assez éphémères, en raison de la rapidité d'évolution des processeurs du marché. De plus, cette architecture dispose d'un espace d'application assez limité, pour lequel le SIMD se simule très bien en programmation SPMD, pour *Single Program, Multiple Data*[9] sur machines à architecture MIMD.

Toutes ces restrictions font que les machines SIMD ont pratiquement disparu du marché des machines parallèles. Nous verrons par la suite que cette architecture se prêtait pourtant assez bien à notre problématique : l'implantation de réseaux de neurones [Clary et Kothari, 1991].

4.2 Les machines MIMD

Les machines MIMD, pour *Multiple Instructions Multiple Data*[10], sont, avec les SIMD, la seconde grande famille d'architectures parallèles "généralistes". Contrairement aux architectures SIMD, ces architectures sont composées d'un nombre plus réduit de processeurs, processeurs de même technologie que les processeurs séquentiels classiques du marché, donc séquentiellement performants. L'idée générale de ces architectures n'est donc plus la conception d'un nouveau type d'ordinateur, différent des architectures séquentielles "Von Neumann". Il s'agit ici de mettre en réseau des processeurs séquentiels pour mettre en commun leur puissance de calcul. La puissance de ce type d'architecture parallèle est donc essentiellement basée sur la puissance des processeurs qui composent la machine.
La famille des machines parallèles de type MIMD se divise, à son tour, en plusieurs architectures, différenciées par l'architecture de la mémoire et les accès des processeurs à celle-ci. Il est ainsi possible de distinguer les architectures à mémoire distribuée des architectures à mémoire partagée, cette dernière catégorie comprenant à son tour les machines de type SMP et les architectures DSM. Nous présentons brièvement ces différentes architectures dans ce chapitre afin d'appréhender leurs spécificités.

4.2.1 Les architectures à mémoire distribuée

Les architectures à mémoire distribuée, aussi appelées "architectures en clusters" sont basées sur une architecture en fait assez éloignée des architectures classiques. Dans ce type d'ordinateurs, l'architecture du réseau de processeurs est divisée en nœuds, chaque nœud du système disposant de son propre espace mémoire (voir figure 4.2). Un nœud peut être composé d'un processeur ou d'une carte de plusieurs processeurs disposant d'une architecture de type SMP (voir page 56), mais la mémoire partagée n'est que très rarement exploitée, au niveau logiciel, entre les processeurs d'un même noeud. La description la plus simple, bien que légèrement schématique, de cette architecture serait de la décrire comme un réseau de machines séquentielles.

La présence de mémoire dédiée pour chacun des processeurs rend cette architecture efficace pour la programmation de type *parallélisme de données*, méthode consistant à donner à chaque processeur la même tâche à effectuer, en distribuant les données à traiter sur les mémoires locales des différents processeurs[11]. Sur ce type d'application parallèle un processeur peut utiliser les variables qui lui sont dédiées sans risque de perturber l'exécution d'autres processeurs. De plus,

9. Programmation dans laquelle chaque processeur exécute une copie du même programme
10. Instructions Multiples sur Données Multiples
11. Une exécution SPMD d'un réseau de neurone consisterait à exécuter le même réseau sur chacun des processeurs utilisés, mais à chaque fois pour traiter un ensemble de données différent.

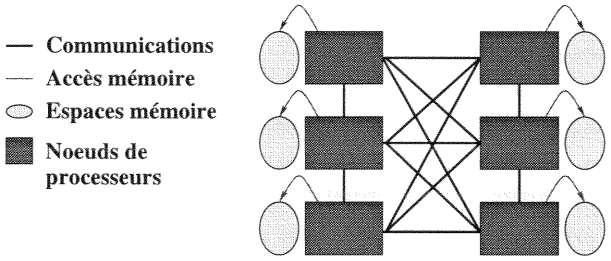


FIG. 4.2 – *Représentation simplifiée d'une architecture à mémoire distribuée. Cette architecture se compose de plusieurs nœuds connectés selon une topologie déterminée. Chacun des nœuds dispose de son espace mémoire propre et peut être composé de plusieurs processeurs.*

l'espace mémoire dédié étant physiquement proche, les temps d'accès aux variables sont courts. S'il s'agit de nœuds comprenant plusieurs processeurs, le nombre de processeurs reste très faible, il n'y a donc pas de risque de goulot d'étranglement (voir page 56). De plus les variables n'étant pas partagées par les différents processeurs du nœud, il n'y a pas de difficultés liées à la cohérence de celles-ci. Pour simplifier la description du modèle, nous décrirons l'architecture en confondant nœuds et processeurs.

Le mode classique de programmation sur ces architectures parallèles est la communication par *envois de message entre les différents processeurs.* S'il existe des interfaces simulant la programmation par *partage de mémoire*, elles ne sont que des sur-couches de protocoles d'envois de messages, leur utilisation entraîne donc une perte de performances.

Les difficultés de programmation proviennent des échanges de données entre les différents processeurs. Lorsque l'application nécessite le partage d'une variable entre différents processeurs, il faut établir des communications entre ces processeurs. Il est nécessaire de construire explicitement les messages au niveau du processeur possesseur de la variable et des processeurs demandeurs de celle-ci. Les temps de latence et de communication, bien qu'en baisse constante, étant très élevés en comparaison des capacités calculatoires de ces machines, il convient de limiter le nombre de messages échangés, et donc de construire des messages contenant un maximum de variables de préférence à une somme de messages contenant chacun une variable.

De même, chaque synchronisation nécessite l'échange de plusieurs messages entre les processeurs à synchroniser, ce qui ralentit encore l'exécution. Par contre l'utilisation de la communication par envoi de messages rend les implantations de ces synchronisations d'un abord assez simple (voir section 3.3.1).

Pour pallier le manque, relatif, d'efficacité de ces machines en termes de communication, le réseau de connexion des processeurs est construit selon des topologies spécifiques, en grille le plus souvent mais aussi en anneau ou en hypercube. Ces topologies ont pour but d'optimiser les communications entre les processeurs et de réduire les temps d'échange de messages, en facilitant le routage des informations ou en réduisant la distance physique entre deux processeurs. Il est ainsi possible, en optimisant les placements des tâches sur les processeurs en fonction du nombre de messages échangés entre elles, d'améliorer les performances d'une application.

Ces architectures à mémoire distribuée s'adaptent donc particulièrement à des applications pouvant être découpées en sous-blocs indépendants et ne nécessitant que peu de synchronisations. De plus, une application développée avec cette sémantique à l'avantage d'être *scalable*, les performances obtenues en termes d'accélération se conservent assez bien lors de l'ajout de

processeurs utilisés pour l'exécution. Ce type de programmes permet de garder potentiellement de bonnes accélérations sur de grands nombres de processeurs.

En revanche, l'obtention de bonnes performances sur ce type d'architecture s'avère difficile en raison des déplacements de données entre les différents nœuds du système et des protocoles de communication à mettre en place pour obtenir cette sémantique d'envoi de messages. La programmation sur ce type d'ordinateurs, en raison de l'utilisation de l'envoi de messages, est éloignée de la programmation séquentielle classique.

4.2.2 Les architectures à mémoire partagée

La mémoire cache

Avant d'aborder plus précisément la description des architectures parallèles à mémoire partagée, attardons-nous un peu sur une spécificité technologique jouant un rôle important dans le développement de cette technologie comme dans l'utilisation de celle-ci, la mémoire cache. Nous présentons dans cette partie les principales définitions et spécificités concernant la mémoire cache qui seront reprises dans la suite de ce manuscrit.

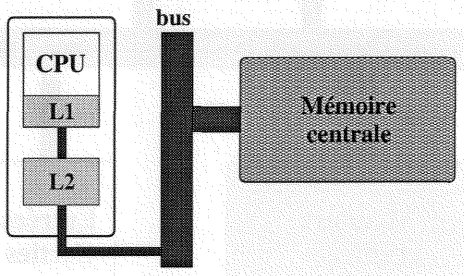


FIG. 4.3 – *Exemple de processeur disposant de deux niveaux de mémoires caches, L1 et L2*

La mémoire cache peut se présenter comme sur la figure 4.3. Cette figure représente une carte avec son processeur (CPU) disposant de deux niveaux de cache, un premier (L1) sur le processeur, et un second (L2) sur la carte. Lorsque le processeur recherche une variable, il scrute son cache de premier niveau, puis, en cas d'absence de la variable, son cache de second niveau. Ensuite seulement, en cas de besoin, il réalise un accès mémoire. L'avantage de la mémoire cache, par sa proximité et sa technologie différente de la mémoire principale, en est la rapidité de son accès aux données. Une donnée présente en *L1* sera atteinte plus rapidement qu'une donnée placée en *L2*, elle-même plus rapidement accessible qu'une donnée placée en mémoire principale.

Dans le cas des architectures parallèles, la présence d'une variable dans le cache exonère le processeur d'un accès à la mémoire par l'intermédiaire d'un bus. Le trafic sur le bus est donc réduit par la présence des caches, ce qui augmente l'efficacité des accès mémoires restants. Nous verrons par la suite qu'un bus surchargé est coûteux en temps d'accès et donc en temps d'exécution.

Les données présentes dans les mémoires caches sont des copies de données ayant leurs adresses physique en mémoire principale. Afin de limiter les transferts de données entre la mémoire et les caches des processeurs, les données ne sont pas copiées une à une mais par blocs. Selon les architectures, les copies peuvent se faire par *pages* ou par *lignes*.
Les protocoles de gestion des caches ont pour but d'y placer les données auxquelles le processeur accède le plus souvent. Mais cette notion de copie signifie, dans le cadre des architectures parallèles, qu'une variable peut être présente dans les caches mémoires de différents processeurs

simultanément, ce qui entraîne la présence d'un protocole assurant la cohérence des données, et donc un certain nombre de communications entre les processeurs en cas de modification d'une donnée, pour les en "prévenir".

La suite montrera qu'une bonne gestion de la mémoire cache influence fortement les performances parallèles d'une application sur certaines architectures.

◇ Les architectures SMP

Les architectures SMP, pour *Symmetric Multi-Processing* [12], sont historiquement les premières architectures MIMD. Elles datent des années 60-70. Le terme *Symétrique* signifie que chaque processeur de la machine perçoit les différentes ressources du système de manière équivalente. Ces ordinateurs sont des architectures à mémoire partagée dotée d'un unique espace de mémoire globale (voir figure 4.4).

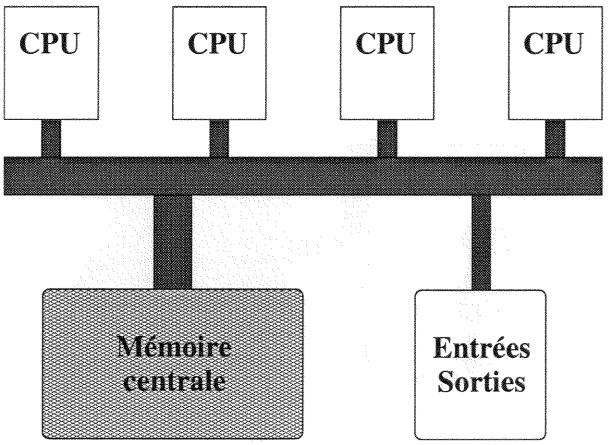


FIG. 4.4 – *Architecture SMP. Chaque processeur est relié par un bus aux différentes ressources de la machine. L'espace mémoire est unique et accessible par chacun des processeurs de la machine parallèle.*

Ces architectures sont très proches des machines séquentielles, la différence étant le nombre de processeurs reliés au bus, qui leur donnent accès à la mémoire commune. Chaque processeur dispose d'un ou de plusieurs niveaux de cache, deux le plus souvent. Le mode principal de communication entre les processeurs est la communication par mémoire partagée [13]. Les accès aux données communes sont synchronisés par la pose de verrous, par le programmeur, protégeant des mises à jour simultanées, i.e chaque variable est accessible en écriture par un seul processeur à la fois.

Ces architectures apportent l'avantage d'une programmation proche de la programmation mono-processeur. Les synchronisations peuvent être relativement intuitives, par utilisation du verrouillage des variables. La mémoire partagée est un système rapide et, l'architecture utilisant des technologies proches de celle des stations de travail séquentielles classiques, ces machines sont bon marché.

Les limites des SMP Le problème de ce modèle d'architecture parallèle est l'accès à la mémoire centrale. L'augmentation du nombre et de la puissance des processeurs a rendu très lourde

12. Machines à multitraitement symétrique
13. Les protocoles d'envoi de messages sont aussi disponibles sur ce type d'architectures

la charge de connexion entre la mémoire centrale et les processeurs (plus de données qui transitent et la fréquence des demandes par processeur qui augmente). Voici donc la limite du modèle : trop de processeurs et d'accès mémoire créent des *goulots d'étranglement*, et ce malgré la présence de caches conséquents sur les processeurs. Les temps de communication deviennent trop importants par rapport aux temps de calcul sur une architecture comportant un grand nombre de processeurs. De plus, chaque augmentation du nombre de processeurs nécessite des modifications technologiques, comme l'augmentation de la bande passante, pour espérer conserver les performances.

Les machines parallèles à architecture de type SMP sont aussi appelées UMA pour *Uniform Memory Access* [14], pour signifier que tous les accès mémoires ont le même coût en temps, que tous les processeurs sont à la même distance de la mémoire, ce qui n'est plus le cas pour les architectures MIMD à mémoire partagée distribuée.

◇ Les machines parallèles à mémoire partagée distribuée

Les deux technologies d'architecture parallèle MIMD déjà présentées, mémoires partagées et mémoires distribuées, disposent donc de caractéristiques assez distinctes [Papadopoulo, 1997; Lefèvre, 1997] qui peuvent se résumer de la manière suivante :

Les architectures à mémoire partagée bénéficient d'une programmation aisée, proche de la programmation séquentielle et de communications transparentes pour l'utilisateur. Cependant le nombre de processeurs efficacement intégrables est limité par la centralisation des accès mémoire.

Les architectures à mémoire distribuée sont très *scalables* et ont l'avantage de permettre des implantations très efficaces. Cependant elles souffrent d'une grande difficulté de programmation, qui limite d'ailleurs souvent leur utilisation au monde de la recherche.

Il était donc légitime d'imaginer une architecture intermédiaire qui unifierait les domaines de compétence de ces deux modèles : une architecture qui permettrait une certaine facilité de programmation, ouvrant la voie à une plus grande variété d'utilisateurs, sur des ordinateurs comprenant un très grand nombre de processeurs, proposant ainsi des gains de temps conséquents, pour inverser le rapport temps de programmation/gains en exécution.

Partie de la mémoire distribuée, inspirée de la mémoire partagée, ainsi fut imaginée la **machine parallèle à mémoire partagée distribuée**, plus couramment appelée DSM [15]. D'abord développé par Kai Li en 1986 [Li, 1986a; Li, 1986b], le concept a séduit et a été développé pour devenir la référence actuelle en matière d'ordinateurs parallèles *non dédiés* [Priol, 1997].

Nous présentons dans cette partie les caractéristiques les plus importantes de ces architectures parallèles.

L'avantage de la DSM sur le concept de la mémoire partagée, est la possibilité d'allier la facilité relative de programmation des machines à mémoire partagée et l'exécution sur un très grand nombre de processeurs [Lefèvre, 1997].

14. Accès Mémoire Uniforme
15. Distributed Shared Memory

◇ Mémoire partagée avec cache

Le problème des machines à mémoire partagée d'architecture SMP est la difficulté de disposer d'un grand nombre de processeurs, et ce en raison de l'activité du bus de communication entre la mémoire centrale et les différents processeurs. L'arrivée de la mémoire cache locale, au milieu des années quatre-vingt, a permis d'envisager une solution : les données sont toujours physiquement placées sur une mémoire partagée, mais lors de l'utilisation d'une donnée par un processeur, cette donnée est copiée dans le cache local de celui-ci. Cette présence réduit fortement le temps de communication ultérieur du processeur pour accéder à cette variable tout en allégeant la bande passante du bus commun.

Il restait néanmoins à garantir aux processeurs la cohérence des données présentes sur le cache, i.e garantir qu'aucun autre processeur n'avait modifié les données entre l'opération de sa copie sur le cache et sa lecture par le processeur.

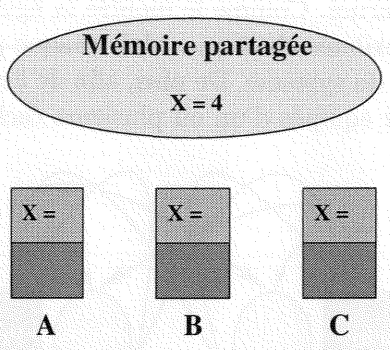
Dans ce but, les premiers systèmes mis en place n'autorisaient qu'une copie par donnée sur les caches de la machine complète. Une donnée utilisée par un processeur était copiée sur son espace de mémoire cache et la donnée était bloquée jusqu'à la fin de son utilisation [Jul *et al.*, 1988; Chase *et al.*, 1989]. Un processeur désirant traiter cette même donnée devait attendre sa libération, la copier à son tour sur son propre espace de mémoire cache et annuler la précédente copie tout en bloquant, à son tour, l'accès à cette donnée.

Les performances obtenues par ce protocole de gestion des caches mémoires ne sont souvent que très peu satisfaisantes. Si plusieurs processeurs désirent lire une même donnée, chacun doit attendre de pouvoir la charger sur son cache, et donc attendre que les autres processeurs aient fini de l'utiliser. Sur ce genre de problèmes, les performances peuvent être moins bonnes qu'une exécution séquentielle (perte due aux temps de communication entre les processeurs et aux temps de copie sur les différents caches).

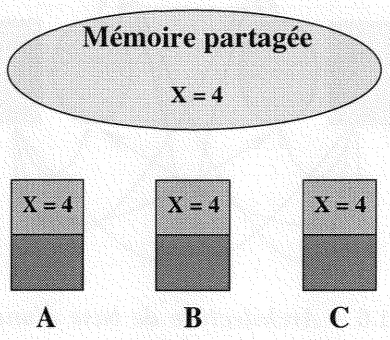
Ces machines autorisent donc une programmation aisée, car proche de la programmation séquentielle, sur un plus grand nombre de processeurs que les architectures SMP classiques, tout en garantissant le déterminisme des applications. Néanmoins, l'obtention de bonnes performances sur ces architectures nécessite de garantir un minimum d'accès consécutifs à de mêmes données pour des processeurs différents et donc un découpage et une gestion des donnés identiques à la programmation sur mémoire distribuée. Une bonne programmation nécessite toujours une importante adaptation des algorithmes séquentiels, ce qui s'avère souvent complexe (voir section 4.2.1).

Afin de palier ces problèmes de performances dus à l'unicité de la copie dans les caches pour les données, des architectures permettant de placer les données sur plusieurs caches simultanément, donc à la dispositions de différents processeurs, furent développées. Avec cette approche, les difficultés de cohérence des données dans le cache, couramment appelée *cohérence de cache* [Chaiken, 1990] sont apparues. La figure 4.5 présente un exemple de problème de cohérence de cache.

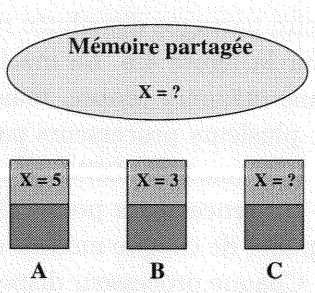
Une nouvelle difficulté liée à la gestion des données partagées est apparue : la possibilité de copies d'une même donnée sur le cache de plusieurs processeurs simultanément n'est pas compatible avec la légitime exigence de *consistance séquentielle* [Lamport, 1979]. Cette propriété réclame en effet que toutes les exécutions d'un même programme doivent être identiques à l'exécution de chacun des processeurs à la suite, elle-même devant être identique à l'exécution séquentielle de ce même programme.
L'obtention de cette consistance séquentielle entraîne la nécessité de la présence d'un protocole de cohérence de cache pour les machines comprenant des processeurs dotés de mémoire cache.



(a) X est en mémoire partagée



(b) A, B et C récupèrent X



(c) A et B font varier X

FIG. 4.5 – *Exemple du problème de cohérence de cache.*
La variable X est une variable partagée, elle est récupérée par les processeurs A, B et C. A pour l'incrémenter, B pour la décrémenter, C pour la lire. Chaque processeur est doté d'un niveau de cache.

◇ Les architectures DSM : les machines parallèles NUMA

Afin de permettre la réalisation d'une architecture parallèle comprenant un nombre élevé de processeurs bénéficiant d'une mémoire partagée, les machines parallèles à *Mémoire Partagée Distribuée*, ou DSM[16] sont apparues. Comme le montre la figure 4.6, ces ordinateurs bénéficient de l'architecture à mémoire distribuée classique. Mais chaque processeur dispose d'un accès à la mémoire des autres processeurs du système. De plus, afin de limiter les transferts entre mémoires et processeurs, ces derniers sont équipés d'un ou plusieurs caches mémoire.

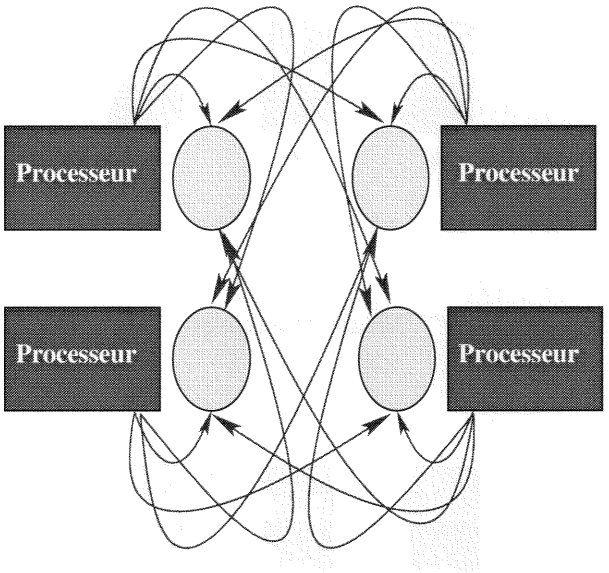


FIG. 4.6 – *Architecture de base d'une DSM.*
Chaque nœud dispose de l'accès à la mémoire des autres nœuds du système.

En fait les architectures à mémoire partagée distribuée regroupent un ensemble plus vaste de machines que le modèle présenté sur la figure 4.6. La machine peut être composée non pas de plusieurs processeurs disposant d'une mémoire propre, mais de plusieurs nœuds, chacun de ces nœuds pouvant à son tour contenir plusieurs processeurs partageant un même espace mémoire.

D'un point de vue technique, une machine DSM peut être vue comme un ensemble de machines parallèles de types SMP (bi-processeurs pour l'Origin 2000, quadri-processeurs pour d'autres architectures). Mais la mémoire de chaque module est intégrée à l'espace d'adressage de chacun des processeurs du système. Chaque processeur dispose de l'accès à une mémoire partagée globale composée de l'union des mémoires de chacun des modules composant le système. Pour le processeur, la seule différence entre les accès à son propre bloc mémoire ou au bloc mémoire d'un module différent est le temps nécessaire à cet accès [Papadopoulo, 1997], ces machines sont à *accès mémoire non uniforme*. Contrairement aux machines à mémoire partagée classiques, dans lesquelles l'accès aux données se fait par un même bus, ici une variable placée sur la mémoire directe d'un processeur sera plus proche qu'une autre placée sur la mémoire d'un processeur différent. Les temps d'accès aux données seront donc dépendants de la distance de la donnée au processeur désirant la traiter. Le rapport entre l'accès distant et l'accès local est appelé *facteur NUMA*[17]. Il est de l'ordre de quatre ou cinq sur les machines actuelles.

16. Distributed Shared Memory
17. NUMA pour Non Uniform Memory Access

Du point de vue de l'utilisateur et du programmeur, ces machines peuvent être vues et programmées, au premier abord, comme des machines SMP. Néanmoins, pour optimiser les temps d'exécution d'une application sur ces architectures parallèles, il reste nécessaire de placer au mieux les données sur la mémoire afin de réduire ce problème de facteur NUMA.

La gestion du cache La recherche de performances inhérente à la technologie parallèle entraîne la recherche de réduction optimale des temps d'accès aux données en mémoire, et donc l'utilisation de mémoire(s) cache(s). Comme nous l'avons constaté précédemment, l'utilisation de cache sur des architectures multiprocesseurs entraîne des problèmes de cohérence des données. La technologie parallèle à mémoire distribuée est donc agrémentée de *protocoles de cohérence de caches* permettant l'utilisation de mémoire partagée distribuée, à l'aide de processeurs dotés de cache(s), tout en préservant la consistance séquentielle, et donc le déterminisme des applications.

Les protocoles de cohérence de cache peuvent être divisés en deux grands types différents [Lefèvre, 1997], *la diffusion* et *l'invalidation*. Ces deux méthodes se distinguent par leur protocole de gestion des mises à jour des données, c'est à dire par leur gestion des écritures en mémoire. Avec la première méthode, la diffusion des écritures, toute modification est immédiatement envoyée à tous les processeurs/nœuds possédant une copie de la donnée. En revanche, l'invalidation des écritures n'autorise la modification d'une donnée par un processeur que lorsque celui-ci est processeur exclusif de la donnée. Des exemples de ces deux grandes familles de protocoles se trouvent dans [Kermarrec, 1996]. Si la diffusion limite le nombre de communications entre les différents nœuds, communications destinées à l'invalidation des copies dans les différents caches, elle augmente le trafic entre les nœuds, pour remettre à jour toutes les copies. Les méthodes à invalidation sont privilégiées par les constructeurs.
Il existe, de plus, différents *modèles de cohérence* dans la classe des protocoles de cohérence de caches [Kermarrec, 1996]. La distinction entre les différents modèles est inhérente à la cohérence du modèle : les modèles à *cohérence forte ou séquentielle* sont les modèles les plus intuitifs. Ils garantissent au programmeur une exécution identique à une exécution séquentielle.
Les modèles à *cohérence faible* ont l'avantage d'accélérer les applications, les accès mémoire entraînant moins d'appels système. Mais ils complexifient la programmation en laissant une partie de la gestion de la cohérence des données au programmeur [Kermarrec, 1996]. Le lecteur pourra trouver des exemples de protocole de cohérence faible dans [Mosberger, 1993; Briggs et Dubois, 1986].

Nous ne présenterons dans la suite que des modèles à cohérence forte, ou séquentielle, qui s'avèrent être les modèles les plus fréquemment rencontrés sur les architectures modernes.

Exemple de protocole de cohérence de cache Il existe de nombreux protocoles de cohérence de cache à forte cohérence basés sur l'invalidation. Ces différents algorithmes ont néanmoins le squelette identique suivant :
Lorsqu'un processeur a besoin d'une donnée, il copie préalablement cette donnée sur son propre espace de mémoire cache. Il dispose ensuite de cette donnée, en lecture, à volonté. Les opérations de lecture s'effectuent comme un accès cache de processeur séquentiel classique, sans aucun besoin de vérification de la validité de la donnée.
La modification d'une donnée, quant à elle, nécessite plusieurs étapes :

1. Le processeur souhaitant modifier la variable rend préalablement non valide les différentes copies de la donnée concernée se trouvant sur les caches d'autres processeurs.
2. Le processeur modifie la donnée dont il dispose maintenant en exclusivité.

3. Les processeurs souhaitant utiliser la variable doivent aller en récupérer une copie valide, qu'ils stockent de nouveau dans leur cache.

Différents algorithmes plus précisément détaillés sont décrits dans [Zhang et Yan, 1995] ou [SGI, 1998].

Cette implantation de protocole de cohérence de cache gère la cohérence au niveau des opérations d'écriture, opérations d'écriture qui génèrent des opérations de communication entre les différents processeurs et leurs caches mémoires, tandis que la lecture s'effectue directement, comme sur une machine séquentielle. Ce mode de cohérence de cache est donc coûteux en écriture. C'est la modification des variables qui pénalise les performances en bloquant des variables. Avant de modifier une variable le processeur doit demander et attendre réception de l'invalidation des diverses copies sur cache de cette donnée. De plus, les processeurs désirant accéder à cette donnée en lecture simple devront attendre la fin de cette modification avant de recharger la donnée dans leur cache propre et de pouvoir ainsi la manipuler.

Afin de réduire le temps nécessaire aux opérations d'accès mémoire, ces protocoles de cohérence de cache sont gérés par le système de la machine parallèle et utilisent les spécificités topologiques de celle-ci afin d'accélérer les communications entre les différents nœuds de celle-ci. Les topologies sont construites afin de permettre une liaison entre les différents processeurs en un nombre limité de communications, en particulier lors des opérations d'invalidation des données.

◇ Classification des architectures

L'introduction de la notion de NUMA permet une nouvelle classification des machines parallèles. La famille des machines parallèles MIMD à mémoire partagée peut se décomposer en trois types de systèmes de mémoire NUMA en termes de migration et de cohérence des données [Zhang et Yan, 1995].

cc-NUMA Pour *cache coherence Non Uniform Memory Access*[18]. Chaque nœud de processeurs dispose d'un cache et d'une partie de la mémoire partagée globale. Le système dispose d'un protocole de cohérence de cache.
La recherche de performances autorise l'utilisation simultanée d'une même variable par plusieurs processeurs en lecture. La modification des variables partagées entre différents processeurs est coûteuse.

Architectures Non cc NUMA Cette appellation fait référence à deux types d'architectures : celles ne proposant pas de cache pour les processeurs et celles fournissant un cache local qui supprime le partage des données et résout ainsi le problème de leur cohérence.
Leur programmation, pour obtenir de bonnes performances, est proche de la programmation pour architectures à mémoire distribuée. Les données doivent être correctement distribuées sur les différents processeurs.

COMA Les architectures parallèles de type COMA, pour *Cache Only Memory Architecture*[19] sont, selon les références, une classe des modèles à mémoire partagée[20] ou une sous-classe du cc-NUMA. La particularité de cette architecture parallèle MIMD est qu'elle ne propose pas d'espace

18. Mémoire à accès non uniforme avec cohérence de cache
19. Architecture uniquement à mémoire cache
20. Comme les classes UMA, cc-NUMA et mémoire virtuellement distribuée.

mémoire physique classique, chaque processeur ne dispose que de mémoire cache. L'ensemble des données est partagée sur les différents caches de la machine et ne dispose pas d'espace d'adressage fixe. Néanmoins, chaque processeur de la machine peut disposer d'une copie de la donnée. De plus, le processeur propriétaire de la donnée, celui détenant la dernière mise à jour, varie au cours de l'exécution. Ce système dispose d'un protocole de cohérence de cache similaire au cc-NUMA. L'un des problèmes de cette architecture réside dans la localisation des données [Priol, 1997]. Pour accéder à une donnée, un processeur doit chercher, parmi tous les processeurs, celui qui dispose de la copie la plus à jour. Par contre, l'avantage principal de ce protocole est qu'il offre un plus grand espace mémoire aux applications [Stenström *et al.*, 1992].

Il n'existe que peu de machines parallèles basées sur cette technologie. Il est toutefois possible de citer la *Data Diffusion Machine*, [Warren et Haridi, 1988] qui fut l'une des premières développées et la KSR [Burkhardt *et al.*, 1992] qui fut l'une des rares à être commercialisées [Priol, 1997].

La mémoire virtuellement partagée Il convient de citer, dans la famille des machines parallèles à mémoire partagée, les mémoires virtuellement partagées ou *MVP*. Contrairement aux modèles de mémoires partagées vus précédemment, la mémoire partagée n'est plus gérée par le matériel mais par une mise en œuvre logicielle. Il est donc question ici de mémoire partagée implantée sur des machines MIMD à mémoire distribuée, et simulée à l'aide des protocoles de communication classiques d'envois de messages disponibles sur ces architectures. Plusieurs modèles existent, le lecteur trouvera une description plus détaillée dans [Kermarrec, 1996] ou [Lefèvre, 1997].

Ce paradigme se voulait une réponse aux difficultés de programmation inhérentes à la communication par envoi de messages sur architecture MIMD à mémoire distribuée. Le développement des architectures à mémoire partagée par la plupart des constructeurs d'ordinateurs parallèles a diminué l'intérêt premier des MVP. Néanmoins la possibilité de simuler de la mémoire partagée pour utiliser des réseaux de stations de travail en tant que calculateur parallèle relance l'intérêt de ce protocole de communication. En effet, cette pratique se développe en raison d'un coût moindre et de la disponibilité du matériel, les stations sont souvent nombreuses dans les sociétés. De plus, les réseaux d'interconnexions deviennent de plus en plus efficaces, permettant des communications performantes entre les différentes stations de travail ; on peut ajouter une souplesse de renouvellement inenvisageable pour les machines parallèles.

Les MVP doivent donc être portables de machines parallèles à stations de travail et pouvoir gérer des architectures hétérogènes sur ce dernier type de réseau de processeurs. Elles peuvent ainsi apporter la simplicité de programmation à mémoire partagée sur ce type de réseau distribué et faire de ces réseaux de véritables calculateurs parallèles.

4.2.3 Conclusion sur les MIMD

La technologie MIMD est à l'heure actuelle la plus présente sur le marché des ordinateurs parallèles. Cette suprématie est essentiellement due au fait que cette architecture permet la construction de machines puissantes en termes de capacité de calculs, contenant un nombre conséquent de processeurs basés sur une technologie proche de la technologie séquentielle, donc d'un coût relativement bon marché. L'apport de la programmation par mémoire partagée entre les processeurs confère à cette architecture une attractivité supplémentaire.

Néanmoins l'obtention de performances sur ce type de machines parallèles nécessite la prise en compte des spécificités technologiques de ces machines, à savoir une forte capacité de calcul

des processeurs, mais des communications et des synchronisations entre ceux-ci beaucoup moins performantes. La puissance de ces machines est essentiellement due à la puissance de calcul de leurs processeurs. S'il existe actuellement différents protocoles de communication sur ces ordinateurs parallèles, la communication par partage de mémoire et la communication par envoi de messages entre les processeurs, la philosophie de la programmation reste dépendante de ce déséquilibre de performances entre puissance de calcul et efficacité des communications. Ce déséquilibre rend les architectures parallèles MIMD beaucoup plus adaptées à la programmation à gros grain qu'à la programmation à grain fin. Cette dernière peut être implantée mais pâtira du nombre conséquent d'échanges de données entre les processeurs.

Une programmation efficace s'avère donc complexe, nécessitant un découpage des instructions en limitant au mieux les synchronisations et les communications entre les processeurs, en prenant garde à l'équilibrage de charge et au placement des données.

4.3 Conclusion

L'univers et le futur des calculateurs parallèles est aujourd'hui essentiellement basé sur les architectures MIMD, le futur semblant privilégier les architectures à mémoires partagées distribuées. Ces machines, bien que dédiées aux applications parallèles à gros grain, permettent toutefois de simuler le grain fin en programmation SPMD, elles occupent ainsi le champ de programmation des architectures de type SIMD, actuellement en voie de disparition.

Il reste toutefois que, quels que soient l'architecture et le type de protocole utilisé, le développement d'applications efficaces nécessite une programmation assez spécifique et adaptée à l'architecture parallèle choisie, et ce malgré les avancées scientifiques du parallélisme, tant au niveau technologique que logiciel, qui tendent à rendre l'utilisation, et par là même la programmation, des machines parallèles de plus en plus accessible. Actuellement le passage d'une application séquentielle sur machine parallèle nécessite toujours un fort investissement en temps d'adaptation des algorithmes. L'implantation des algorithmes parallélisés devient, quant à elle, de plus en plus simple avec l'arrivée de nouveaux outils comme OpenMp, sur machines parallèles MIMD à mémoire partagée. Les principales difficultés d'adaptation sont la gestion des synchronisations, des communications entre différents processeurs, l'équilibrage de charge et, surtout, la gestion et le placement des données utilisées par l'application.

L'accès à cette technologie, pour un non spécialiste, avec pour seul objectif l'accélération de ses applications nécessite donc toujours un intermédiaire, humain ou logiciel.

Chapitre 5

Simulations et parallélisme des réseaux connexionnistes

Introduction

La famille des réseaux de neurones artificiels peut se diviser en deux classes, la classe des modèles statistiques et la classe des modèles d'inspiration biologique.
Les premiers utilisent les architectures classiques évoquées au cours des chapitres précédents, les familles des perceptrons multi-couches, des cartes auto-organisatrices ou des modèles de Hopfield. Ces modèles de réseaux sont constitués de neurones formels, binaires ou continus, fortement connectés.
Les seconds s'inspirent plus directement d'observations généralement effectuées par des neurobiologistes, biochimistes ou psychologues, pour définir de nouveaux paradigmes calculatoires. Dans ce cas les unités de base sont beaucoup plus complexes que les neurones formels de McCulloch et Pitts et leurs dérivés. Les différentes fonctions internes de ces unités vont généralement au delà de la fonction d'activation sur une somme pondérée des entrées. Elles comprennent souvent des mises à jour de variables locales au neurone, ses *variables d'état*, ou des protocoles de prise en compte du temps.

Ces deux familles disposent toutefois des mêmes paradigmes calculatoires :

- Une architecture massivement parallèle.
- Un mode de calcul et une mémoire distribués.
- Un grand nombre d'unités.

Une autre des caractéristiques communes aux différents modèles connexionnistes est que les applications engendrées sont connues pour être coûteuses en temps de calcul. D'une manière générale, le mode d'apprentissage des modèles connexionnistes, basé sur une lente convergence du réseau par correction des différents -et nombreux- paramètres de celui-ci, coûte très cher. Un modèle statistique peut nécessiter une très longue phase de mise au point, tant au niveau de la détermination d'une architecture adaptée que de l'apprentissage des nombreux paramètres du réseau. Les réseaux de neurones sont connus pour leur capacité à résoudre de nombreux problèmes, mais aucune théorie ne permet de définir, à partir d'un problème posé, la topologie du réseau correspondant. Le réseau le plus utilisé dans les applications connexionnistes reste le perceptron multi-couches, réseau disposant d'un algorithme d'apprentissage bien connu pour la

lenteur de sa convergence. En plus de l'apprentissage, la configuration d'un perceptron multi-couche est une opération pouvant actuellement prendre plusieurs jours.

Non seulement le connexionnisme statistique est une solution généralement coûteuse en temps de calcul mais, de plus, la taille et la complexité des réseaux développés augmente sans cesse. Les réseaux tendent en effet à s'adapter à des applications réelles toujours plus importantes, nécessitant toujours plus de paramètres. Une autre raison de la croissance des réseaux de neurones artificiels est qu'ils s'intégrent aussi dans des systèmes complexes de traitement des données, systèmes pouvant comporter un grand nombre de réseaux hétérogènes interconnectés.

De la même manière, les modèles biologiques sont coûteux en temps de calcul. Ces modèles sont fréquemment composés d'unités de base plus complexes que le simple neurone formel. Si les unités de bases sont souvent des neurones temporels, la granularité utilisée peut aussi être plus grande, en utilisant comme unité de base les colonnes corticales [Durand, 1995; Frezza-Buet, 1999], unités plus lourdes en exécution que les simples neurones formels. A l'intérieur des unités de base des réseaux, le grain de simulation des phénomènes naturels devient de plus en plus fin. A l'image des neurones de Taylor, les neurones artificiels tentent de s'inspirer au plus près des connaissances sur les réactions électrochimiques intra et extra neuronales.

Afin d'approcher certains protocoles naturels, les modèles biologiques deviennent de plus en plus volumineux en nombre d'unités. En effet, à l'image de leur modèle biologique, le cerveau, les réseaux neuromimétiques travaillent sur des réseaux contenant un nombre très important de neurones. Mais, contrairement au cerveau qui fonctionne sur la base de *population de neurones* faisant émerger les propriétés cognitives, les réseaux de neurones artificiels actuels simulent ces populations algorithmiquement, ce qui biaise d'autant les résultats observés. La suite logique des travaux dans le domaine neuromimétique est maintenant l'implantation des populations de neurones, les réseaux seront donc basés sur de très grandes quantités de neurones, neurones ayant eux-mêmes une très grande connectivité. Des simulations informatiques de réseaux de cette taille sont impossibles, tant au niveau de la puissance de calcul des ordinateurs que de la taille des mémoires disponibles sur les ordinateurs actuels et même à venir dans les prochaines années, et ce malgré la vitesse de la progression technologique.

Cette recherche de puissance de calcul, inhérente à la recherche en réseau de neurones artificiels, entraîne naturellement à s'intéresser aux outils informatiques proposant les meilleures performances dans ce domaine : les calculateurs parallèles.

En plus des capacités proposées par ces machines, le passage au parallélisme, ardu pour la plupart des problèmes, pourrait sembler aisé au niveau connexionniste. Les réseaux de neurones sont en effet connus pour leur parallélisme naturel, dit *intrinsèque*. Exécuter des réseaux d'unités de calcul neuronales sur un réseau de processeurs semble être une solution efficace et simple à appréhender pour répondre aux limites technologiques rencontrées par le connexionnisme.

Nous allons, dans cette partie, présenter un survol des parallélisations des réseaux connexionnistes. Nous allons tout d'abord présenter le(s) parallélisme(s) présent(s) dans les architectures de réseaux de neurones artificiels. Nous verrons ensuite les différentes méthodes utilisées pour implanter différentes architectures spécifiques sur machines parallèles. Puis, avant d'aborder les modèles de simulateurs connexionnistes parallèles, nous présenterons plus précisément les mérites comparés des différents types de simulateurs séquentiels. Ces différents systèmes existant nous serviront à déterminer nos choix pour la conception de notre propre simulateur.

5.1 Le parallélisme intrinsèque des réseaux de neurones

Les réseaux de neurones sont souvent décrits comme des modèles de calcul et de mémoire distribués. Mais au delà du lieu commun, Tomas Nordström [Nordström, 1995; Nordström et Svensson, 1992] propose une étude plus détaillée des parallélismes disponibles au sein des modèles connexionnistes.

Algorithme 5.1 Décomposition d'un algorithme classique de réseaux de neurones. D'après [Nordström et Svensson, 1992]

pour tout Session d'apprentissage **faire**
  pour tout Exemple de la session **faire**
    pour tout Couche du réseau **faire**
      pour tout Neurone de la couche **faire**
        pour tout Poids du neurone **faire**
          pour tout Octet de la valeur du poids **faire**

Partant d'une décomposition d'une généralisation de l'exécution d'un réseau connexionniste, telle que présentée par l'algorithme 5.1, Nordström extrait six principaux degrés de parallélisme dans l'algorithmique connexionniste, chaque degré affinant le parallélisme :

1. *Le parallélisme des sessions d'apprentissage.*
2. *Le parallélisme des exemples d'apprentissage.*
3. *Le parallélisme des couches.*
4. *Le parallélisme des neurones.*
5. *Le parallélisme des synapses.*
6. *Le parallélisme des octets.*

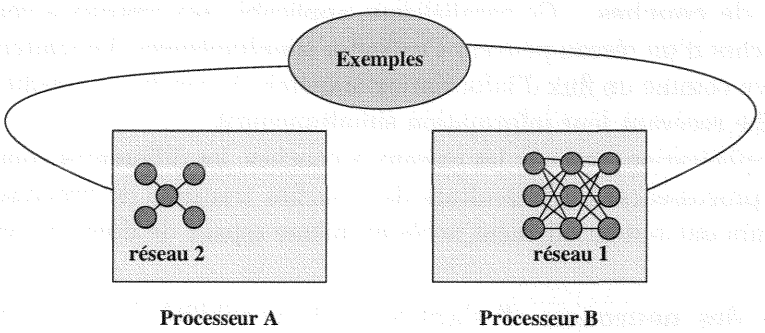


FIG. 5.1 – *Utilisation du parallélisme de session d'apprentissage. Des réseaux différents peuvent être exécutés simultanément, sur des espaces d'exemples différents, en étant placés sur des processeurs différents.*

Le parallélisme de session d'apprentissage. Il est possible d'exécuter et de faire apprendre, simultanément, différents réseaux. Ce degré de parallélisme signifie, concrètement, qu'il est possible de faire apprendre une même tâche à deux architectures neuronales différentes. Ce parallélisme peut être utile, par exemple, pour déterminer la topologie d'un réseau adaptée à un

problème donné, dans le cas d'architectures connexionnistes statiques. Il est ainsi possible de faire apprendre simultanément plusieurs architectures, avec différentes topologies et fonctions d'activations, puis de choisir son architecture en comparant les résultats obtenus sur chacun des réseaux, par exemple pour déterminer le nombre de couches et de neurones par couche d'un perceptron multi-couches.

En termes d'implantation, ce parallélisme peut être schématisé grossièrement comme sur la figure 5.1. Il est possible de placer un réseau de neurone différent sur chaque processeur, et de faire apprendre ces réseaux, simultanément, sur un même corpus d'apprentissage.

Le parallélisme des exemples d'apprentissage. Ce degré de parallélisme signifie qu'il est possible de faire traiter plusieurs exemples différents simultanément à un même réseau.

En terme d'implantation, il s'agit de disperser les éléments du corpus d'exemples et de faire s'exécuter une copie identique d'un même réseau sur chacun des processeurs (voir figure 5.2). Chaque copie du réseau calcule ses sorties et met à jour, différemment, ses poids. Chaque processeur travaille sur un unique espace de données.

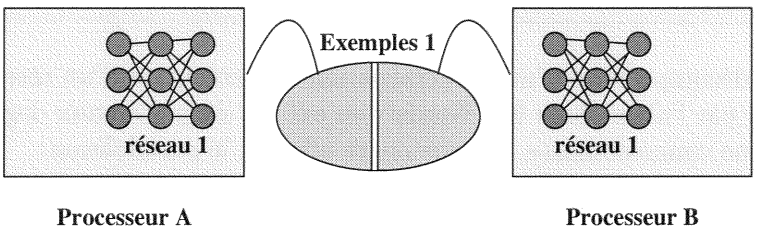


FIG. 5.2 – *Le parallélisme des exemples d'apprentissage. Le réseau est recopié intégralement sur chaque processeur et l'espace des exemples est réparti sur ces processeurs.*

Le parallélisme de couches. Ce parallélisme, applicable aux réseaux à couches, indique que les différentes couches d'un réseau peuvent s'exécuter simultanément. Le traitement d'un réseau à couche peut être vu comme un flux d'information transmis de couche en couche, tous les neurones d'une même couche recevant leur information simultanément.

En terme d'implantation, et pour les réseaux à couches, les différentes couches sont réparties sur les différents processeurs et les résultats des couches transmis de processeur en processeur. Tous les processeurs ont potentiellement accès au même espace de données (voir figure 5.3).

Le parallélisme des neurones. Il s'agit ici de la possibilité d'exécuter simultanément les tâches imparties aux différents neurones d'un même réseau.

En terme d'implantation, les neurones du réseau sont répartis sur les différents processeurs et travaillent de façon concurrente. L'espace des entrées est partagé par les différents processeurs (voir figure 5.4), chaque neurone ayant potentiellement accès aux données.

Le parallélisme des synapses. Ce parallélisme postule la possibilité d'exécution concurrente des différentes synapses d'un même neurone, et par là même, des différentes synapses d'un même réseau.

En terme d'implantation, les différentes synapses d'un même neurone peuvent être réparties sur différents processeurs lorsque le neurone doit effectuer la somme pondérée de ses entrées.

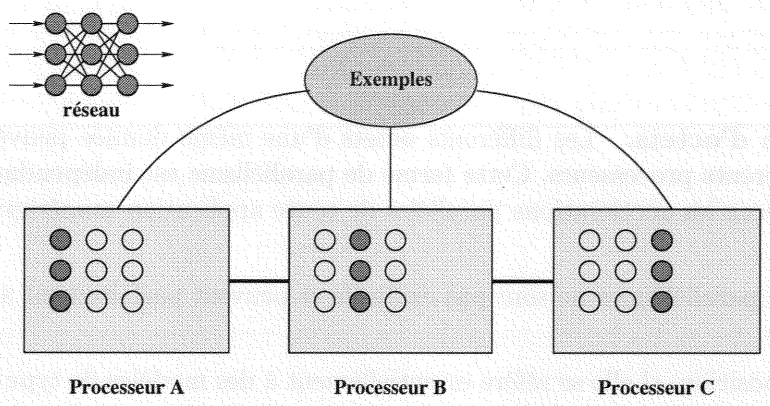


FIG. 5.3 – *Le parallélisme des couches du réseau. Chaque couche est placée sur un processeur distinct. Les différents couches du réseaux travaillent donc simultanément.*

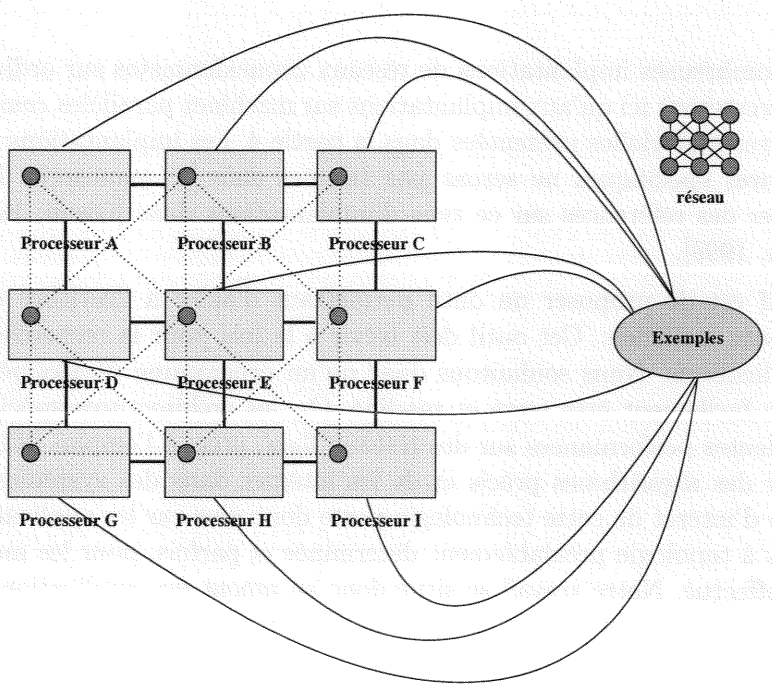


FIG. 5.4 – *Le parallélisme de neurones. Les neurones d'un même réseau sont répartis sur les différents processeurs et peuvent s'exécuter simultanément. L'espace des données est partagé entre les différents processeurs.*

Les différentes pondérations peuvent ainsi être effectuées de manière concurrente, charge est au neurone d'effectuer ensuite la somme résultante.

Le parallélisme d'octets. Les différents octets d'une même donnée peuvent être traités en parallèle sur différents processeurs. Cette forme de parallélisme est indépendante du connexionnisme. Elle concerne les accélérations parallèles de toute application numérique "lourde".

Ces différents parallélismes ne sont pas exclusifs et peuvent parfaitement s'imbriquer les uns dans les autres.

Cette décomposition, si elle se réfère essentiellement à des modèles de type perceptron multicouches à rétropropagation du gradient, peut se généraliser à tout le domaine connexionniste, particulièrement en conditionnant le point 3, le parallélisme de couches.
L'utilisation du parallélisme informatique nécessitant, en préalable à l'implantation, une décomposition du problème en tâches *parallèles*, c'est-à-dire en tâches pouvant être exécutées simultanément, c'est à partir de ce(s) parallélisme(s) inhérent aux modèles connexionnistes que se sont développés les différentes implantations parallèles de modèles connexionnistes effectuées ces dernières années.

S'il existe de nombreuses implantations de réseaux connexionnistes sur ordinateurs parallèles, nous ne nous intéresserons ici qu'aux implantations sur machines parallèles *courantes*, c'est-à-dire sur les architectures matérielles présentées dans la partie 4. Les implantations sur cartes dédiées ou sur architectures spécifiques ne seront pas traitées dans ce manuscrit. Le lecteur pourra néanmoins trouver des références sur ce type d'implantations dans [Girau, 1999; Sundararajan et Saratchandran, 1998].

Notre objectif est de proposer un outil permettant d'aider à l'écriture de simulations de réseaux de neurones artificiels. Cet outil doit servir à la fois pour la recherche et pour le développement d'applications. Nous souhaitons donc qu'un programme développé à l'aide de notre simulateur puisse facilement être testé et modifié. Or, les architectures spécifiques permettent d'obtenir d'excellentes performances sur des réseaux figés, prêts à l'emploi. Elles permettent par ailleurs de câbler des algorithmes précis et de les intégrer dans des systèmes embarqués complexes. Le champ d'intérêt de cette technologie porte donc plus sur les applications des modèles, pour des modèles à topologie préalablement déterminée et parfois, pour les modèles embarqués, à apprentissage effectué. Notre travail se situe donc en amont des applications inhérentes à ces technologies.

Reprennant la décomposition de Nordström, les degrés de parallélisation 1 et 2, de *sessions* et *d'exemples*, permettent des implantations parallèles à gros grain, tandis que les degrés 4 et 5, *de neurones* et *de synapses*, favorisent les implantations à grain fin des réseaux.

Le domaine d'investigation des travaux de parallélisation des réseaux de neurones est vaste. La grande majorité des travaux concerne l'implantation d'une architecture précise, ou d'un apprentissage spécifique, sur une machine cible déterminée. Mais il existe aussi des simulateurs plus *génériques*, ces derniers étant, le plus souvent, des extentions de simulateurs séquentiels.

5.2 Les implantations parallèles de réseaux de neurones spécifiques

Il existe de multiples implantations parallèles de réseaux de neurones à topologie fixe. Ainsi toutes les grandes architectures connexionnistes classiques connaissent des implantations spécifiques [Sundararajan et Saratchandran, 1998], les cartes auto-organisatrices [Demian et Mignot, 1996; Barbosa et Lima, 1990] ou les réseaux de Hopfield [Margaritis et Evans, 1992] par exemple. Une revue bibliographique plus conséquente peut être trouvée dans [Misra, 1996]. Les parallélisations concernent aussi des réseaux à diffusion plus confidentielle et demandant de grandes ressources de calcul. Par exemple, certains réseaux d'inspiration biologique ont été implantés sur différentes architectures de machines parallèles [Quoy *et al.*, 1997; Lallement, 1996; Pican, 1996; Wacquant, 1993].

Toutefois l'architecture sur laquelle se font la grande majorité des travaux dans le domaine reste sans conteste le perceptron multi-couches et son fameux apprentissage par rétropropagation du gradient [Kumar *et al.*, 1994; Petrowski *et al.*, 1993; Petrowski, 1993; Zhang *et al.*, 1990; Chu et Wah, 1992; Östermark, 1996].

Les parallélisations spécifiques sont généralement des implantations parallèles de bas niveau recherchant des performances en terme de temps d'exécution. Ce sont des algorithmes spécifiques développés pour s'adapter au parallélisme proposé par la machine hôte et mis au point par des spécialistes de la programmation parallèle. Pour obtenir de bonnes performances, les décompositions des algorithmes connexionnistes englobent le maximum de parallélisme connexionniste afin d'atteindre les objectifs d'efficacité recherchés.

Une autre stratégie utilisée pour implanter les réseaux connexionnistes sur machine parallèle consiste à utiliser les formalismes mathématiques de bas niveau de ces réseaux [Gégout *et al.*, 1995; Girau, 1995]. D'une manière plus radicale, il est aussi possible de ne considérer les opérations sur les réseaux de neurones que d'un point de vue mathématique et de transformer les exécutions de réseaux en calculs uniquement matriciels. Les différentes machines parallèles construites pour être des super-calculateurs possédant des outils optimisés pour le calcul algébrique, le parallélisme de ces applications est ensuite rapide et peut permettre d'obtenir de très bonnes accélérations. Mais il est difficile de déterminer un protocole étendant l'"algébrisation" d'une architecture à l'ensemble des réseaux statistiques. De plus, les modèles biologiques sont a priori exclus de ces méthodes.

5.2.1 Les méthodes d'implantation sur architectures SIMD

La grande majorité des implantations de réseaux spécifiques sont des implantations sur machines parallèles SIMD, celles-ci étant d'ailleurs presque exclusivement effectuées sur MasPar ou sur Connection Machine.

Ces architectures ont séduit en raison d'une présence et d'une disponibilité supérieure avant l'émergence des nouvelles générations de machines MIMD, mais aussi, et surtout, en raison de propriétés conceptuellement assez proches de celles des réseaux connexionnistes. Elles disposent en effet d'un nombre élevé de processeurs (jusqu'à 4096 pour la MasPar) dotés de communications efficaces, aboutissant à une technologie puisant sa puissance de calcul dans la somme de ses unités et l'efficacité des communications entre ces unités plus que dans la puissance individuelle de chacune des unités.

Pour implanter un réseau connexionniste sur des machines parallèles SIMD, modéliser chaque neurone sur un processeur semble la solution la plus évidente. Ces architectures offrant un nombre suffisant de processeurs, cela peut permetttre des simulations utilisant simultanément les paral-

lélismes de neurones, de synapses et d'exemples. Le réseau peut ainsi utiliser un processeur par neurone, un processeur par connexion et le réseau est recopié sur l'architecture matérielle jusqu'à utilisation du nombre maximal de processeurs.

Néanmoins ce modèle d'implantation parallèle se heurte au grand nombre de communications inhérentes aux modèles connexionnistes. Cette large connectivité des réseaux de neurones (le nombre de connexions est généralement de l'ordre du carré du nombre d'unités) entraîne une débauche de communications entre les différents processeurs. De plus, ces communications sont souvent difficiles à router au niveau de la machine parallèle et s'avèrent peu efficaces en raison des connexions irrégulières des modèles connexionnistes, qui se simulent par des communications entre processeurs distants sur la topologie matérielle [Paugam-Moisy, 1995].

5.2.2 Les méthodes d'implantations sur architectures MIMD

Pour les implantations sur architecture parallèle de type MIMD, il devient impossible, en raison du faible nombre de processeurs disponibles, de dédier un processeur par neurone du réseau. La parallélisation considérée comme la plus *naturelle*, la parallélisation par neurones, devient ainsi impossible à utiliser pour effectuer l'implantation matérielle.

La seconde grande difficulté inhérente à l'utilisation de machines MIMD est conséquence du nombre de connexions présentes dans les réseaux de neurones et de la fréquence des communications à travers celles-ci. Ces propriétés des réseaux de neurones rendent très complexe l'obtention de performances sur ce type d'ordinateurs. Les méthodes de parallélisation utilisées sont essentiellement basées sur les parallélismes de couches et d'exemples.

Le parallélisme de couche permet de *pipeliner* les communications, en limitant les communications entre deux processeurs à un unique message mais sans en changer la fréquence.

Le parallélisme d'exemples permet de limiter à la fois le nombre et la fréquence des communications entre les différents processeurs de la machine, ce qui correspond parfaitement aux conditions permettant d'obtenir des performances sur ce type d'ordinateur.

En termes d'efficacité, le parallélisme d'exemples est le plus performant. Cette méthode d'implantation consiste à copier le réseau de neurones complet sur chacun des processeurs disponibles. C'est ensuite l'espace des données qui est *parallélisé*, en fait divisé en autant de parties que de processeurs disponibles. Ainsi, chaque processeur dispose d'un réseau complet et d'un espace de données pour l'exécuter séquentiellement. Les (nombreuses) communications entre neurones se déroulent localement sur chacun des processeurs, et n'entraînent donc ni de communication entre les différents processeurs ni même de synchronisation entre ceux-ci. Chacun des processeurs peut alors progresser à son rythme sur son espace de données et n'est jamais stoppé par l'attente du résultat de l'un des processeurs voisins. L'un des gros inconvénients de cette méthode est due au coût en mémoire inhérent, le réseau et l'espace des poids et paramètres de celui-ci étant intégralement recopiés dans la mémoire de chacun des processeurs. La mémoire de la machine parallèle occupée par l'application est donc multipliée par le nombre de processeurs utilisés par celle-ci.
En phase d'exécution, cette utilisation du parallélisme d'exemples est la meilleure. Les communications entre les différents processeurs étant nulles, l'accélération est linéaire. En effet, chacun des réseaux est totalement autonome sur l'espace des exemples qui lui est présenté. Cette méthode est ainsi équivalente à l'exécution du réseau sur plusieurs machines séquentielles classiques, sans communication entre elles, et chacune sur son fichier de données. Il n'y a donc pas de parallélisation spécifique dans ce cas.
En phase d'apprentissage, en revanche, l'implantation devient nettement plus complexe. Il est

nécessaire de communiquer régulièrement entre les différents processeurs pour effectuer, pour chacun des poids, une prise en compte générale des multiples apprentissages locaux. L'apprentissage local de différents réseaux entraîne des corrections différentes de chacun des paramètres du réseau en raison d'une présentation d'exemples différents. Il est alors nécessaire de mettre à jour régulièrement chaque poids du réseau parallélisé en tenant compte de chacune de ses copies [Puzenat, 1997; Paugam-Moisy, 1992; Girard et Paugam-Moisy, 1994; Girau et Paugam-Moisy, 1995].

Les communications dues à ces mises à jour limitent alors les performances. Il existe de nombreuses stratégies pour limiter le nombre de rendez-vous de mise à jour durant l'apprentissage. L'une d'elles repose sur la *spécialisation* des espaces de données présentés à chacune des copies du réseau. Le but de cette stratégie est de spécialiser les processeurs sur l'apprentissage de caractéristiques distinctes de l'espace des exemples. Dans le cas d'une carte topologique, chaque processeur ne ferait converger qu'un nombre fini de neurones du réseau.

Cette méthode de parallélisation nécessite donc une connaissance des caractéristiques des exemples d'apprentissage présentés au réseau, un prétraitement de cet ensemble d'apprentissage pour le diviser efficacement et des modifications conséquentes des algorithmes d'apprentissage des réseaux connexionnistes.

Cette méthode est opposée à la philosophie de l'apprentissage des réseaux de neurones, apprentissage basé sur une convergence lente du réseau vers les propriétés décrites par l'espace des données présentées. Cette convergence se fait par l'interaction de toutes les unités du réseau au travers de sa topologie.

Les, nombreux, algorithmes de parallélisation de réseaux par parallélisation des exemples réclament donc, de la part du programmeur, de profondes modifications des algorithmes connexionnistes classiques, modifications qui changent le résultat de l'apprentissage. Il est alors difficile de garantir la qualité des apprentissages obtenus. De plus, ces méthodes de parallélisation entraînent une augmentation conséquente du nombre d'itérations nécessaires lors de l'apprentissage, ce qui limite les accélérations des exécutions.

En plus de l'investissement en temps, ces modifications algorithmiques rendent les solutions en termes de parallélisation peu adaptables d'une architecture connexionniste à une autre.

Le *parallélisme de données*, implanté sous forme de *parallélisme d'exemples* dans le cas des réseaux de neurones, est donc un modèle qui, s'il donne les meilleures performances en terme d'accélération des applications sur les machines parallèles de type MIMD, demande un gros effort d'adaptation de l'algorithme de départ à l'architecture matérielle cible. Les solutions obtenues sont difficilement applicables à d'autres problèmes connexionnistes.

Le problème de ces implantations réside dans leur faible portabilité et leur manque total d'adaptabilité. Développés par des programmeurs spécialistes du parallélisme, et adaptés à leur support matériel d'exécution, ces programmes ne sont généralement pas manipulables par un non spécialiste, disons un programmeur de réseaux de neurones, en vue d'une amélioration purement connexionniste des modèles. Un réseau parallélisé ne permet aux connexionnistes ni de paralléliser d'autres réseaux, ni même de paralléliser une version modifiée d'un algorithme connexionniste déjà parallélisé.

Ces modèles de parallélisation de réseaux de neurones sont donc très éloignés de nos objectifs, la simulation des réseaux connexionnistes en général, et de réseaux neuromimétiques en particulier. Plus proches de notre démarche, il existe des simulateurs de réseaux de neurones sur

machines parallèles.

Avant de nous intéresser aux simulateurs dédiés aux exécutions sur machines parallèles, et pour préciser notre propos sur ceux-ci, nous présentons les simulateurs séquentiels génériques.

5.3 Les modèles de simulation de réseaux de neurones

Les implantations d'architectures connexionnistes spécifiques correspondent peu à notre problématique. Elles proposent des stratégies qui ne sont généralement pas transposables d'une architecture de réseau à une autre et qui ne permettent pas aux utilisateurs du connexionnisme de développer leurs propres modèles sur machines parallèles.

Souhaitant proposer un outil permettant d'implanter toute sorte d'architectures connexionniste et d'utiliser des machines parallèles pour les exécutions, notre problème est donc de développer un simulateur de réseaux connexionnistes permettant des exécutions parallèles.

Il existe actuellement peu de simulateurs de réseaux de neurones pour machines parallèles et ceux qui existent sont, à l'exception notable de Cupit [Prechelt, 1994], des extensions de simulateurs séquentiels.

Par contre, pour ce qui est des simulateurs séquentiels, de nombreux travaux ont été entrepris, et bien souvent abandonnés d'ailleurs. Un très bon état de l'art des simulateurs séquentiels de réseaux de neurones peut être trouvé dans [Kock et Serbedzija, 1996].

Les simulateurs peuvent être classés en trois catégories distinctes, selon les fonctionnalités proposées :

- Interfaces Graphiques.
- Bibliothèques.
- Langages spécifiques.

Les *interfaces graphiques* représentent la catégorie des "clickodrômes", les simulateurs permettant de spécifier les modèles dans des menus préétablis. Les bibliothèques et les langages proposent quand à eux des outils permettant de programmer les modèles, à l'aide d'un langage de programmation courant pour les *bibliothèques*, à l'aide d'un langage totalement dédié au connexionnisme pour les *langages spécifiques*.

5.3.1 Les interfaces graphiques de simulation

Les interfaces graphiques à menus sont dédiées à l'utilisation rapide et conviviale de modèles connexionnistes classiques, au niveau de la topologie comme des algorithmes d'apprentissage. Ils proposent des implantations optimisées de ces modèles et permettent les combinaisons classiques entre ceux-ci. Leur gros avantage réside dans la facilité d'approche : pas de langage informatique, mais une interface classique et ergonomique utilisable à la souris et au clavier (voir figure 5.5) .

Le plus connu de ces outils est SNNS [Zell et al, 1993] mais il en existe d'autres comme NeuralWorks [NeuralWare, 1989] ou NeuroGraph [Wilke *et al.*, 1995]. A partir des menus, l'utilisateur est invité à construire et à spécialiser son application. Pour ce faire, il peut spécifier les différents paramètres de son réseau : nombre de couches, nombre de neurones, type de neurones, topologie du réseau, algorithme d'apprentissage, type d'apprentissage, etc.

Ces systèmes fournissent en général de nombreux outils pour construire les réseaux, mais aussi pour analyser et contrôler les exécutions. Pour construire les réseaux, ces simulateurs s'appuient essentiellement sur des éléments de base prédéfinis. L'utilisateur choisit dans les divers menus les

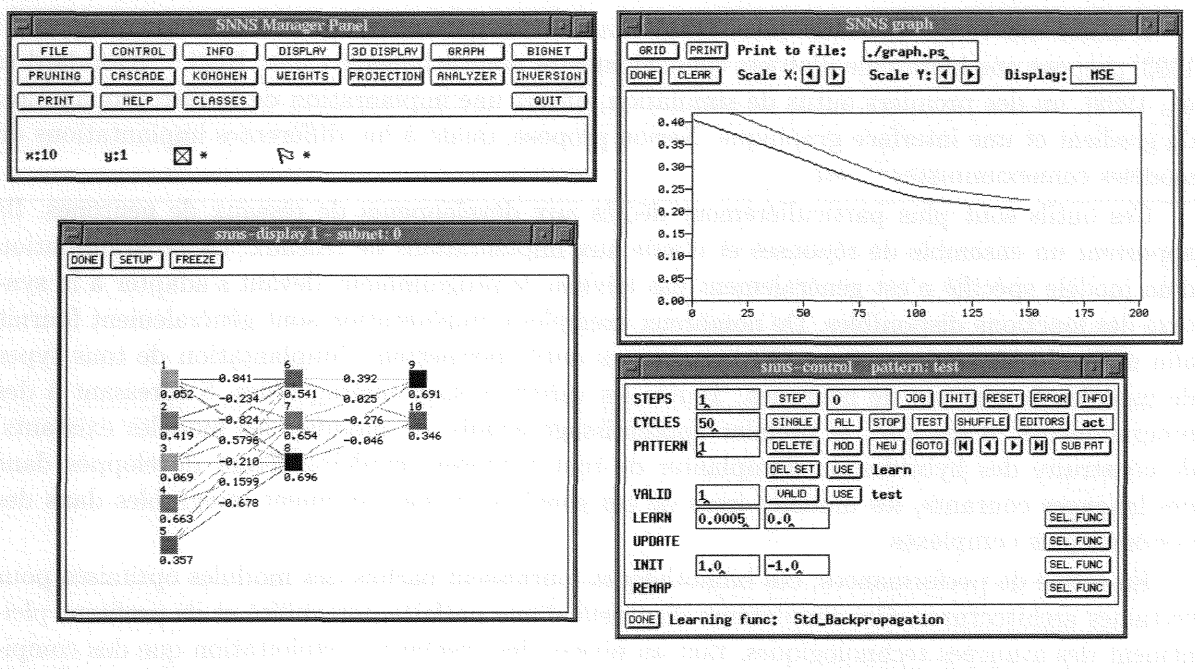


FIG. 5.5 – *Un exemple d'interface graphique de simulation, SNNS.*

éléments composant son réseau parmi les modèles prédéfinis, au niveau de l'architecture comme au niveau des différents algorithmes utilisés.

Si ces outils sont performants lorqu'il s'agit de mettre en œuvre des architectures connues, de les tester sur des données propres, et d'en modifier les différents paramètres, leur flexibilité est beaucoup moins satisfaisante. Avec ces outils, il est souvent extrêmement difficile de construire ses propres réseaux, de mettre au point ses propres fonctions d'activation ou de spécifier les caractéristiques temporelles de ses neurones. Lorsqu'elle existe, la construction de ces spécificités passe par l'écriture de son propre matériel à l'aide de langages dédiés, difficiles à manipuler. Cette implantation de nouveaux outils demande une connaissance approfondie du simulateur, de son implantation et de la structure de ses données afin de pouvoir les modifier.

En terme d'efficacité, ces simulateurs offrent en général d'assez bonnes performances, puisqu'ils se présentent comme optimisés pour les modèles proposés. Ils supportent donc généralement les réseaux de grande taille. Pour tenter d'améliorer ses performances, SNNS propose une possibilité d'exécution sur machine SIMD, MasPar [Zell *et al.*, 1993].

Ces simulateurs sont plus dédiés à l'utilisation des réseaux de neurones qu'à la recherche en connexionnisme. Ils facilitent l'usage des réseaux plus que l'implantation proprement dite de nouveaux modèles ou protocoles.

5.3.2 Les bibliothèques de simulation

Cette deuxième grande famille d'outils de simulation repose sur un ensemble de fonctions informatiques reposant sur un langage de programmation classique, généralement le C ou le C++.

Il existe différentes bibliothèques, ayant chacune ses particularités. Sesame[21] [Linden *et al.*, 1993] propose une hiérarchie d'objets connexionnistes pour le langage C++. RCS,[22] [Goddard *et al.*, 1989], un des premiers outils de simulation, offrait une implantation de la rétropropagation du gradient et une interface graphique. Xerion propose, quant à lui, différentes implantations de modèles connexionnistes[23], etc.

Ces outils sont plus particulièrement dédiés aux développeurs de réseaux de neurones. Ils apportent un ensemble de réponses et d'aide aux implantations de réseaux. La programmation d'un modèle spécifié n'est généralement pas triviale, le programmeur devant s'adapter à la syntaxe des fonctions disponibles. De nombreux exemples d'implantation sont généralement fournis afin de l'aider dans cette tâche. Néanmoins ces outils permettent l'implantation de tous types de modèles de réseaux de neurones. Toutes les variantes sont envisageables. S'adressant à des programmeurs, il est aisé, le premier apprentissage acquis, de modifier les modèles existants, de construire des hybrides ou d'implanter de tout nouveaux modèles. Etant développés dans des langages courants, les modèles issus de ces simulateurs sont aisément intégrables dans des systèmes plus complexes.

En terme de performances, ces bibliothèques fournissent parfois des modules optimisés pour certaines architectures. Ces simulateurs disposent d'une parfaite portabilité et ils profitent pleinement des avancées technologiques, tant au niveau des systèmes d'exploitation que des compilateurs et des microprocesseurs.

5.3.3 Les langages spécifiques de simulation

La dernière grande famille des simulateurs génériques connexionnistes concerne les systèmes comme Aspirin [Leighton, 1992], PlaNet [Miyata, 1991] ou AXON [Hecht-Nielsen, 1990]. Ce sont des langages spécifiques de programmation dédiés aux modèles connexionnistes et permettant aux programmeurs de ce domaine de définir tout ou partie de leurs réseaux. Ils peuvent généralement être enrichis de bibliothèques de spécification. Aspirin dispose, par exemple, de son interface graphique, appelée Migraine.

Le plus souvent, ces simulateurs sont basés sur un concept utilisé ensuite pour formaliser et construire les modèles connexionnistes. Ces langages peuvent être plus ou moins généralistes, ils peuvent être conçus pour développer un unique type de topologie, ou tous les types de réseaux répondant ou non au concept fondateur.

Certains langages comme DESIRE [Korn, 1995] sont spécialisés dans la modélisation mathématique des réseaux de neurones. Dans ce cas, toutes les opérations neuronales sont traitées comme des opérations de matrices et de vecteurs, les aspects plus connexionnistes (les concepts de neurones, de synapses, de communication, de synchronisation, etc.) n'interviennent que pour d'éventuelles représentations graphiques.

D'autres langages, comme AXON, proposent un concept radicalement différent. Ce langage s'appuie sur un neurone formel générique et propose de construire des topologies à partir de celui-ci. A l'image de son grand frère biologique, l'unité de base du modèle ne tolère qu'une unique valeur en sortie. Cette *radicalité* du modèle limite énormément ses potentialités. La simple simulation d'une rétropropagation du gradient, si chère au cœur de la congrégation, nécessite de multiples stratégies de détours pour la phase rétropopagée de l'apprentissage [Kock et Serbedzija,

21. Software Environment for the Simulation of Adaptive Modular Systems
22. Rochester Connectionist Simulator
23. Backpropagation, Recurrent Backpropagation, Boltzmann Machine, Mean Field Theory, FEM, CL, Self-Organizing Map, Learning Vector Quantizer

1996]. Ainsi, les poids et les activations se simulent sous la forme de neurones ce qui éloigne la topologie finale du modèle d'origine et complexifie à l'extrême la phase de développement.

Certains simulateurs, CONNECT [Kock et Serbedzija, 1994] ou GENESIS [Wilson *et al.*, 1990] par exemple, proposent un langage de type objets. Ils permettent de générer des classes de neurones, de synapses, etc. pour ensuite en faire hériter les objets du réseau. Ici, la syntaxe est souvent très proche du C++. Les classes de neurones sont définies en spécifiant les variables d'entrées, de sorties, et les différentes opérations sur ces variables. Ce type de simulateur peut proposer des langages allant du plus simple, comme CONNECT où une grande partie de l'implantation se fait en programmation classique, au plus complexe, comme GENESIS qui propose un langage à part entière avec sa syntaxe propre et de multiples fonctions dédiées et complexes.

Ces langages sont, après un sérieux apprentissage, généralement simples à utiliser, du moins pour des modèles répondant strictement aux concepts pour lesquels ils ont été développés. S'il est aisé de construire des modèles à couches à l'aide des langages mathématiques, leur utilisation pour développer un modèle cortical s'avère beaucoup plus problèmatique.

L'intégration de modèles développés à l'aide de ce type de simulateur dans un système plus complexe ne pose, en théorie, aucun problème, les langages offrant souvent la possibilité de générer du code C ou C++.

En terme de performances, les simulateurs ne recherchent pas tous les mêmes objectifs. Si les langages mathématiques recherchent généralement les performances en terme de temps d'exécution, ce n'est pas toujours le cas des langages plus conceptuels. Certains, comme CONNECT, font le choix de proposer un outil essentiellememt axé sur l'aide à l'implantation des réseaux sans optimisation particulière. D'autres, comme GENESIS, cherchent à optimiser les temps d'exécution des réseaux répondant à leur *concept*. Ces langages s'avèrent beaucoup plus complexes à manipuler [Plonski, 90] en raison d'une syntaxe extrêmement minutieuse permettant les optimisations à la compilation.

Pour améliorer les performances et permettre l'implantation et l'exécution de réseaux de grande taille, certains langages permettent une exécution sur ordinateur(s) parallèle(s). Les transferts sur machines parallèles sont plus ou moins transparents. Ainsi, CONNECT propose une implantation sur machines MIMD très *naïve*, mais transparente, des réseaux à couches [Kock *et al.*, 1996] et GENESIS offre une version parallèle du langage permettant des exécutions sur machines parallèles, en mettant à la disposition de l'utilisateur des commandes spécifiques afin de lui permettre de paralléliser ses applications [PGENESIS, 2000].

En terme de concepts connexionnistes, les simulateurs permettent en général de décrire principalement des modèles de type statistique et moins les modèles *biologiques*. Actuellement la recherche dans le domaine connexionniste travaille beaucoup sur la prise en compte du temps dans les modèles, et très peu de simulateurs permettent de descendre à ce niveau de détail des modèles. Le plus avancé de ces simulateurs est GENESIS, qui permet de décrire précisément les modèles d'un point de vue explicitement biologique. Ainsi il est possible de décrire précisément les activités du soma, de l'axone ou des dendrites en explicitant le rôle et les équations d'activation de chacun de ces éléments. Ce degré de précision des modèles implique un simulateur doté d'un langage très complet, et par là même complexe à dominer et ne permettant que des exécutions de petits modèles, en raison du coût des calculs [Strey, 1999]. Mais GENESIS est un langage plus précisément dédié aux neurobiologistes qu'à la communauté connexionniste.

5.3.4 Discussion sur les différentes formes de simulateurs généralistes

En conclusion, il existe de nombreux outils permettant de simuler les réseaux connexionnistes. Il reste à l'utilisateur à faire son choix. Les différents types de simulateurs ont sensiblement les mêmes potentialités en terme de capacité d'intégration, de possibilité à être intégrés dans un système plus complexe, de *scalabilité*, de possibilité à gérer des réseaux de grandes tailles. L'intégration, qui ne pose pas de difficultés pour les bibliothèques de fonctions, se fait, pour les interfaces graphiques et les langages spécifiques, par leur capacité à générer du code C ou C++. Mais aucune garantie n'est apportée sur la qualité de ce code généré.

Le choix du type de simulateur dépend donc essentiellement des besoins de l'utilisateur.

Si l'utilisateur souhaite obtenir rapidemment des résultats à partir d'un modèle classique utilisant des algorithmes tout aussi classiques, l'utilisation des interfaces graphiques est la meilleure solution. Ce type de simulateur lui donnera aisément accès aux modèles classiques, sans connnaissances particulières en connexionnisme, ni même en programmation informatique. Il pourra aisément tester la solution connexionniste sur un problème donné. Les architectures seront claires et souvent optimisées. C'est la solution idéale pour l'utilisateur de réseaux de neurones.

En ce qui concerne la recherche sur le domaine, les deux autres solutions sont préférables. Les avis sont partagés sur les qualités comparées de ces deux types de simulateurs. Kock et Serbedzija, dans [Kock et Serbedzija, 1996], estiment les langages spécifiques préférables, une fois l'apprentissage de ceux-ci effectué par l'utilisateur. Les outils disponibles avec les langages spécifiques sont plus riches, et souvent plus souples à utiliser, si le réseau développé correspond au concept neuronal choisi par le langage. Dans ce cas, la syntaxe du langage et les outils se trouvent en totale adéquation avec les besoins de l'utilisateur.

De plus, les langages présentent le grand interêt de permettre de compiler le code écrit par l'utilisateur. Cette possibilité leur permet d'obtenir des syntaxes relativement conviviales, et de pouvoir optimiser certaines exécutions, ce qui peut être très efficace, notamment dans le cas d'exécutions parallèles.

Si les langages proposent généralement une plus grande variété d'outils que les bibliothèques, c'est essentiellement dû au fait qu'un outil absent d'un langage ne peut être trouvé ailleurs. Un langage doit donc fournir un kit complet pour pouvoir être utilisé de manière satisfaisante. Une bibliothèque, par contre, ne propose que les outils pour lesquels elle apporte une aide en terme d'implantation ou de performance, les autres outils et fonctions restent fournis par le langage de base de ladite bibliothèque, ou par les outils annexes à ce langage. Par exemple une bibliothèque développée sur les langages C ou Fortran disposera implicitement de tous les outils disponibles sur ces langages : debuggers, librairies graphiques ou mathématiques, profilers, etc. et ce sur toute plateforme technologique disponible. Ainsi un outil manquant à une librairie peut être disponible par le biais d'autres outils sur le langage de développement, ou peut être programmé par l'utilisateur dans ce langage.

Pour les mêmes raisons, une bibliothèque disposera des nombreux travaux effectués sur le langage choisi comme référence. Par exemple, une bibliothèque développée sur le langage C ou C++ aura accès aux nombreux outils développés sur et pour ce langage par la communauté GNU, ainsi qu'aux compilateurs optimisés par les constructeurs sur toutes les architectures.

Pour sa part, un langage compilé ne béneficiera pas d'une recherche aussi poussée sur le code généré et ne sera, au mieux, qu'une surcouche d'un langage classique, avec un surcoût au moins comparable à celui d'une bibliothèque. Un langage de performances, les langages mathématiques

par exemple puisqu'ils génèrent du code machine, nécessite un effort important et constant de portabilité de la part de ses concepteurs, effort qu'il est difficile de garantir à l'utilisateur.

La maintenance est en effet l'un des problèmes principaux des langages spécifiques, et ce quel que soit le domaine d'utilisation de ces langages. Pour exister, un langage demande un suivi en terme d'adaptation aux nouvelles architectures et technologies informatiques, par exemple aux nouvelles puces, aux nouveaux systèmes d'exploitation, aux nouvelles cartes graphiques. Au rythme des nouveautés dans le domaine informatique, cet effort est considérable. Par exemple, que devient un langage non porté sur Windows NT ou Linux?

Ainsi un programme, de réseau de neurones par exemple, écrit dans un langage spécifique sera totalement perdu pour ses utilisateurs si le langage est abandonné par ses concepteurs. Le travail effectué, tant au niveau de la conception de nouvelles architectures connexionnistes et d'algorithmes que des efforts d'adaptabilité au langage, à ses mots clé, sa syntaxe et son concept, se trouve complètement perdu en cas d'abandon du langage. Or, les langages étant généralement développés par des universitaires à titre expérimental, les abandons sont très fréquents. Ainsi en cas de besoin d'un langage mathématique de simulation de réseaux, il est plus prudent d'utiliser la surcouche connexionniste offerte par un logiciel mathématique de grande distribution comme Matlab [Demuth et Beale, 1993] plutôt qu'un langage plus spécialisé, plus simple à utiliser car proposant des outils plus adapté, mais n'assurant aucune garantie de suivi.

5.4 Les simulateurs parallèles

Dans leur grande majorité les simulateurs parallèles sont des extensions de simulateurs séquentiels, comme par exemple les extensions de CONNECT ou de SNNS. Des simulateurs spécifiquement développés pour décrire des modèles connexionnistes au sens large et permettre d'obtenir des performances sur machines parallèles sont très peu nombreux.

CuPit CuPit est probablement le plus abouti de ces simulateurs. Il propose un langage spécifique très complet sur machine SIMD [Prechelt, 1994; Prechelt, 1999], permettant de décrire les réseaux et d'obtenir ainsi de bonnes performances sur MasPar. La description des réseaux se fait sous forme d'objets ***neurones***, ***connexions*** et ***réseau***, le simulateur utilise ensuite le parallélisme de neurones pour effectuer l'implantation sur machine parallèle.

La raréfaction du parc de MasPar, et de machines parallèles SIMD en général, a conduit à l'écriture d'un nouveau langage spécifique, CuPit2, permettant des exécutions sur machines parallèles MIMD de type SMP. Ce nouveau langage propose à l'utilisateur, avec une description identique des réseaux, de choisir son degré de parallélisme entre les parallélismes de neurones, connexions ou exemples. L'utilisation du parallélisme d'exemple est fortement conseillée pour obtenir des performances sur architectures SMP. Mais, si le nombre de répliques du réseau peut être géré par le langage, les modifications algorithmiques permettant les mises à jour des poids en fonction des différents apprentissages locaux sont à la charge du programmeur [Hopp et Prechelt, 1997; Hopp et Prechelt, 1999].

Pour obtenir de bonnes performances sur machines parallèles de type MIMD, le programmeur doit donc modifier fortement ses algorithmes pour les adapter au parallélisme de données.

PGENESIS est une version parallèle du langage GENESIS, langage permettant de décrire très précisément les différentes fonctionnalités des réseaux de neurones artificiels. C'est un langage peu utilisé par la communauté connexionniste, il est plutôt dédié aux descriptions des modèles

biologiques. Ce langage permet d'effectuer des simulations sur machines parallèles supportant PVM, mais la parallélisation (répartition des neurones, synchronisations, etc.) est explicitement décrite par l'utilisateur. Le parallélisme n'est donc pas masqué, les fonctions sont juste simplifiées [PGENESIS, 2000]

EspiloNN est un langage permettant de simuler des réseaux de neurones et de les exécuter uniquement sur machines parallèles SIMD. Ce langage, dédié aux implantations de réseaux mathématiques sous forme de matrices et de vecteurs, a ensuite été étendu à l'implantation de modèles plus biologiques. Mais les architectures potentiellement simulables avec cet outil sont limitées par l'impossibilité de développer des réseaux évolutifs [Strey, 1997; Strey, 1999].

Thilo Reski Dans son mémoire de doctorat, [Reski, 1999], Reski propose un début de langage de simulation de réseaux de neurones sur machines parallèles de type MIMD. Les applications développées utilisent la communication par envoi de messages entre les différents processeurs. Pour l'exécution, les neurones sont regroupés par blocs, blocs contenant les éléments (neurones) ayant les mêmes entrées, les mêmes sorties et les mêmes fonctions d'activation. Les éléments d'un même bloc sont regroupés sur le même processeur (ou sur plusieurs processeurs dans le cas de blocs de grande taille). Ce regroupement par blocs permet de limiter les communications entre les processeurs en regroupant les communications entre les différents neurones sur un nombre limité de messages entre les processeurs.

L'utilisation d'un langage permet d'obtenir les informations nécessaires à ces regroupements par blocs au cours d'une analyse du réseau effectuée lors de la compilation. Cette méthode nécessite la connaissance des caractéristiques du réseau, c'est-à-dire du nombre de neurones et de la topologie du réseau, à la compilation et donc en début d'exécution. Cette méthode ne permet donc pas l'implantation de réseaux évolutifs : le nombre des neurones comme celui des connexions est fixé. De plus, du fait du regroupement par blocs, le nombre de modèles potentiellement performants est relativement faible.

Une interface parallèle plus générale : ParCeL-1 Pour développer des réseaux de neurones sur machines parallèles, il est aussi possible d'utiliser une interface avec la machine, interface non spécifiquement dédiée au connexionnisme, dans le but de simplifier la phase de programmation sur machines parallèles. Nous avons, dans l'équipe Cortex, souvent utilisé le langage ParCeL-1 [Cornu, 1992; Vialle, 1996; Vialle *et al.*, 1998] pour paralléliser certains de nos modèles [Lallement, 1996]. ParCeL-1 est un langage spécifique, servant d'interface pour la programmation parallèle à envoi de messages sur architectures MIMD. C'est un langage de programmation parallèle dédié aux applications d'intelligence artificielle, implanté sous forme de langage d'acteurs synchrones.

Ce langage permettait, entre autre applications, d'implanter les réseaux de neurones, en utilisant le parallélisme de neurone. Il permettait de programmer les réseaux au niveau des neurones, en explicitant formellement le code d'exécution du neurone et les connexions des différents neurones. Au cours des applications les neurones échangeaient donc explicitement les informations.

ParCeL-1 était basé sur une communication à envoi de messages, distribuait les neurones implantés sur les différents processeurs et gérait pour l'utilisateur les communications entre les processeurs (en s'appuyant sur les communications entre les différents acteurs).

Ce langage, s'il facilitait la parallélisation des réseaux de neurones, se heurtait au grand nombre et à la grande fréquence d'échanges de messages entre les neurones. La recherche de performances obligeait le programmeur à limiter et optimiser les communications, et donc à abandonner le parallélisme de neurones pour un parallélisme moins naturel (regroupement des

neurones) pour tenir compte de contraintes liées à la programmation sur machine parallèle.

5.5 Conclusion

Il n'existe donc que très peu de simulateurs parallèles de réseaux de neurones, et la plupart d'entre eux sont des langages proposant des exécutions sur machines parallèles de type SIMD. Cette technologie permettait en effet d'utiliser le parallélisme de neurones (et parfois de synapses) en exécutant un neurone par processeur. Néanmoins la recherche de performances demande une connaissance de la topologie du réseau ce qui limite souvent, à l'exception de CuPit, les modèles simulables à des modèles statiques [Prechelt, 1999].

La quasi disparition des architectures SIMD a, semble-t-il, stoppé la recherche dans ce domaine. Les différences entre une programmation efficace sur machine MIMD, c'est à dire une programmation limitant fortement les communications entre les différents processeurs, et les contraintes liées au connexionnisme rendent les implantations performantes extrêmement complexes. La meilleure méthode d'implantation passe par le parallélisme des exemples, méthode impliquant de profonds changements au sein des algorithmes de réseaux de neurones, changements algorithmiques peu portables d'une architecture connexionniste à une autre. Il semble donc extrêmement difficile de proposer un simulateur effectuant une parallélisation transparente aux connexionnistes. Des méthodes concurrentes, et passant par des communications par envois de messages, comme celle proposée par Reski, rendent les modèles potentiellement performants très rares et imposent des limitations fortes sur les architectures connexionnistes implantables.

Si la suprématie de la technologie MIMD des machines parallèles semble avoir stoppé les implantations parallèles des réseux de neurones, les besoins en puissance de calcul des modèles connexionnistes restent bien réels. L'arrivée nouvelle des architectures de type mémoire partagée distribuée (DSM), qui proposent des protocoles de programation par partage de mémoire efficaces sur un grand nombre de processeurs, dans les technologies matérielles peut donc constituer l'occasion d'un renouveau dans le domaine croisé du parallélisme et du connexionnisme.

Deuxième partie

Une bibliothèque de simulation parallèle des réseaux connexionnistes

Chapitre 6

Fondements et développements

Après avoir présenté les domaines des réseaux de neurones artificiels, du parallélisme informatique et des simulateurs connexionnistes, nous présentons dans ce chapitre les objectifs recherchés pour notre simulateur et les choix qui nous ont guidés dans sa construction.

Ce chapitre est donc notre synthèse commentée et critique de la partie précédente du manuscrit. Ayant exposé et commenté les différents aspects des domaines dans lesquels notre étude s'inscrit, nous pouvons maintenant justifier nos choix de développement en fonction des conclusions obtenues sur ces différents aspects.

6.1 Les choix de développement

Nous souhaitons proposer un simulateur permettant aux connexionnistes de développer facilement leurs modèles. Avec ce simulateur, les connexionnistes doivent pouvoir implanter des réseaux de grande taille et effectuer leur exécution sur les machines parallèles les plus répandues. S'adressant à des informaticiens, mais pas à des spécialistes du parallélisme, notre simulateur doit rendre l'implantation parallèle *transparente*. L'utilisation de notre simulateur pour la programmation des réseaux connexionnistes ne doit en rien dépendre des contraintes liées à la programmation parallèle. De plus notre simulateur souhaite faciliter l'implantation des réseaux de neurones sans demander à l'utilisateur de gros efforts d'adaptation.

Pour ce faire nous nous proposons de développer une bibliothèque sur le langage C, utilisant le parallélisme *intrinsèque* des réseaux de neurones, le parallélisme de neurones, pour faciliter la programmation des réseaux et rendre possible les exécutions sur machines parallèles MIMD.

6.1.1 Le choix de la machine cible

Nous avons choisi de développer notre simulateur sur deux types de machines cibles, les ordinateurs parallèles de types MIMD à mémoire partagée et les ordinateurs séquentiels classiques.

Le choix des machines parallèles

Notre outil doit permettre d'accélérer les exécutions de réseaux connexionnistes et de simuler des réseaux de neurones de grande taille, composés d'un grand nombre d'unités et de connexions. Pour ce faire nous avons opté pour la solution consistant à utiliser le parallélisme matériel. Actuellement, et en raison des choix techniques, cette technologie se développe et devient plus aisément accessible. De grands centres de calcul existent qui mettent à la disposition de la

communauté scientifique ce type de super-ordinateurs[24]. En utilisant le réseau Internet, il devient ainsi possible d'utiliser le parallélisme informatique sans contraintes infrastructurelles, le réseau rendant possible l'utilisation de machines parallèles géographiquement lointaines.

Ces raisons justifient notre choix de nous investir sur les machines parallèles du *marché*, les machines *généralistes*, plutôt que d'investir dans une technologie matérielle spécifique, comme les neuro-ordinateurs, qui impose un investissement financier (achat du matériel supporté par la seule communauté connexionniste) et risqué, rien ne garantissant la pérénité des travaux effectués ni même celle de la technologie choisie.

Ces choix "politiques" effectués, le choix du type de technologie parallèle, le choix entre les machines de types MIMD et celles d'architecture SIMD, est plus rapide.

Au premier abord, la technologie parallèle de type SIMD semble beaucoup plus adaptée à notre problématique. Cette architecture est en effet plutôt dédiée au parallélisme *à grain fin*, qui se caractérise par de multiples exécutions effectuées en parallèle, séparées par de fréquentes opérations de synchronisation et/ou de communication. Or, une représentation grossière d'un réseau de neurones peut être un ensemble de nombreuses unités, à faible potentiel computationnel, effectuant de nombreuses, et fréquentes, opérations de communication entre elles. Cette analogie explique pourquoi la plupart des implantations parallèles ont été effectuées sur machines SIMD, et ce malgré la non régularité des connexions au sein des modèles connexionnistes, non régularité qui rend la plupart des implantations relativement complexes [Paugam-Moisy, 1995].

Malgré ces similarités de granularité, les implantations les plus efficaces de réseaux de neurones sont les implantations sur machines MIMD, architectures plutôt dédiées aux applications parallèles *à gros grain*. Ces implantations sont basées sur le parallélisme de données[25], parallélisme à gros grain des réseaux de neurones, au prix de nombreuses modifications des algorithmes connexionnistes classiques (voir section 5.2.2).

La quasi disparition des machines de type SIMD du parc des machines parallèles (il n'en existe actuellement que peu en activité[26] et les constructeurs semblent avoir totalement abandonné cette technologie pour les raisons évoquées dans la partie 4.1.3), nous a naturellement porté à choisir la technologie MIMD comme machine parallèle cible.

Les machines séquentielles

Notre simulateur a pour ambition de faciliter le travail des chercheurs dans le domaine du connexionnisme. Il doit donc permettre aux utilisateurs de développer et de mettre au point de nouvelles architectures et de nouveaux algorithmes. Il nous a semblé important de permettre aux développeurs d'effectuer leurs mises au point, et les inévitables phases de *débogage* sur stations de travail à architectures séquentielles. De même, la possibilité donnée à l'utilisateur d'effectuer, au choix, ses exécutions sur machines parallèles ou séquentielles permet ainsi d'utiliser notre simulateur pour implanter toutes sortes de réseaux, en gardant l'option parallélisme pour les applications lourdes. L'utilisateur n'aura donc pas à décider de la machine cible avant de développer son modèle, il n'aura pas non plus à récrire ses implantations en cas d'augmentation excessive de la taille de ses réseaux.

24. Comme exemples de centre de calcul, il est possible de citer l'IRISA en Bretagne, le CRIHAN en Haute-Normandie et, bien sûr, le Centre Charles Hermite en Lorraine

25. Rappel : Ces implantations sont faites en recopiant le réseau sur chacun des processeurs utilisés et en distribuant l'espaces des données sur ces processeurs

26. Les centres de calculs évoqués plus haut proposent essentiellement des architectures MIMD : Intel Paragon à l'IRISA, SGI Origin2000 pour le centre Charles Hermite et le CRIHAN

Ce double choix d'architecture permet enfin d'effectuer la phase d'apprentissage d'un réseau sur machine parallèle, puis d'utiliser ensuite le réseau sur toute station de travail. Cette possibilité peut être appréciée pour des modèles qui, comme les perceptrons multi-couches, connaissent une longue phase d'apprentissage ou pour des modèles à élagage, modèles pouvant contenir un très grand nombre d'unités et de connexions en début d'apprentissage, mais pouvant atteindre une taille beaucoup plus raisonnable à la sortie de celui-ci [Bougrain, 2000].

6.1.2 Le choix du degré de parallélisme neuronal : le concept neuromimétique

L'idée d'apporter la solution du parallélisme matériel aux difficultés de performances des réseaux de neurones vient essentiellement du *parallélisme intrinsèque* de ces modèles. Les réseaux connexionnistes sont en effet considérés comme des modèles parallèles et distribués.
Nous souhaitons donc tout à la fois conserver les propriétés parallèles des modèles connexionnistes et les utiliser pour les implantations sur machines parallèles.

Souhaitant développer un simulateur permettant d'implanter tout à la fois des réseaux de type *statistique* mais aussi des réseaux de type *biologique*, il est impossible, ou en tout cas extrêmement difficile de trouver un modèle mathématique algébrique général permettant la simulation de ces deux types de réseaux (voir section 5.2). Un tel modèle, s'il existait, pourrait ensuite être efficacement et relativement rapidement parallélisé à l'aide des nombreux outils algébriques disponibles sur toutes les architectures matérielles.

Comme nous l'avons vu dans la section 5.1, Nordström et Svensson décrivent, dans [Nordström et Svensson, 1992], six degrés de parallélisme présents dans les modèles neuromimétiques. Pour paralléliser une application il est nécessaire d'en extraire les tâches pouvant être exécutées de manière concurrente. Utiliser le parallélisme inhérent aux modèles connexionnistes implique donc d'utiliser au moins l'un de ces six degrés de Nordström.
De ces six parallélismes, les parallélismes de *sessions d'apprentissage* et *d'octet* ne sont en rien propres au domaine des réseaux de neurones et n'ont que peu d'intérêt pour le domaine. Exécuter des applications sensiblement différentes sur des processeurs, ou même des ordinateurs, différents (parallélisme de session) ne pose aucune difficulté, et paralléliser les calculs (parallélisme d'octet) n'est intéressant que pour des applications lourdes en calcul, ce qui n'est pas le cas des réseaux de neurones.

Le parallélisme *des exemples* est le plus adapté aux architectures matérielles de type MIMD [Paugam-Moisy, 1995], il est aussi le plus utilisé, avec le parallélisme de *couches*, sur ces architectures. Notre simulateur a pour objectif de permettre de développer des réseaux de neurones sans adaptation des algorithmes à la machine choisie pour l'exécution, en utilisant les algorithmes *classiques* des réseaux. Or le parallélisme d'exemples nécessite de grandes modifications algorithmiques. Le parallélisme de couches, qui nécessite lui aussi des modifications, limite son application aux réseaux à couches.

Le parallélisme *de neurones* est le plus intuitif. Les réseaux de neurones sont, en effet, essentiellement représentés comme un ensemble de neurones, fonctionnant indépendamment et déterminant leurs activités en fonction de leurs entrées.
Le parallélisme *de synapses* est beaucoup moins naturel. Le plus souvent, le rôle des synapses est simulé au niveau des neurones propriétaires de ces synapses. L'utilisation de ce parallélisme impliquerait donc des modifications des algorithmes pour distinguer explicitement les rôles des

synapses du rôle des neurones. De plus ce degré de parallélisme neuronal implique des implantations à grain très fin, chaque synapse ayant une part très faible dans la somme de calculs effectuée par un réseau, grain peu adapté au parallélisme MIMD.

La contrainte d'un simulateur utilisable sans modification algorithmique contraint donc au choix de l'utilisation du parallélisme de neurones pour l'implantation parallèle, bien qu'a priori peu adapté à l'architecture cible choisie. Ce degré de parallélisme nous semble le plus naturel pour les utilisateurs, le plus proche de la sémantique connexionniste. En effet, la description habituelle d'un modèle connexionniste passe par une description algorithmique du type de neurones utilisés, de la topologie liant les neurones, et de l'algorithme permettant d'effectuer l'apprentissage de ce modèle.

Nous souhaitons proposer un simulateur permettant aux utilisateurs de décrire des modèles plus proches des modèles biologiques que des modèles statistiques. Le programmeur devra envisager son réseau comme un ensemble d'unités de base[27], fonctionnant en parallèle. Il doit décrire la tâche effectuée par les unités de base du réseau, tâche dépendant uniquement de leurs entrées et de leurs, éventuelles, variables locales, et permettant de définir la sortie de cette unité de base. C'est la topologie, les connexions entre les différentes unités de base, qui doit faire émerger les propriétés attendues du réseau.

6.1.3 Le type de simulateur : une bibliothèque de fonctions

Comme nous l'avons vu dans la partie 5.3, il existe différentes classes dans les simulateurs de réseaux de neurones : les boîtes à outil sous forme d'interfaces graphiques, les langages de programmation dédiés aux réseaux de neurones et les bibliothèques de fonctions sur un langage de programmation courant.

Souhaitant proposer un simulateur permettant de développer, outre les architectures classiques, toutes sortes d'architectures et d'algorithmes alternatifs, la forme "Interface Graphique" n'est pas adaptée. Ce type de simulateur est principalement dédié à l'utilisation de réseaux préalablement implantés dans le simulateur.

Notre simulateur se propose d'aider les chercheurs en connexionnisme à implanter leurs modèles. Par là-même, il s'adresse à des programmeurs expérimentés, mais dont l'activité principale est consacrée aux problèmes liés au connexionnisme. Notre outil doit donc être le plus simple possible à utiliser tout en apportant de nombreuses fonctionnalités pour le développement.

Si, comme nous l'avons vu dans la partie 5.3.3, un langage permet d'offrir des outils optimisés assortis d'une syntaxe conviviale, il implique, de la part de l'utilisateur, un investissement pour acquérir une connaissance suffisante du langage, de sa syntaxe et de ses mots clefs, et lui permettre de développer ses propres applications. Cet investissement est relativement risqué au vu de la courte durée de vie de la plupart de ces langages. Il est d'ailleurs possible de remarquer que, quel que soit le domaine informatique concerné, peu de langages sont utilisés, et ces langages sont les grands langages généralistes.

Afin de pouvoir offrir un outil attractif, et portable, nous avons opté pour un simulateur sous forme de bibliothèque de fonctions sur un langage courant.

27. A priori des neurones, mais rien n'interdit d'implanter des réseaux décrits à partir de colonnes corticales ou de maxi-colonnes.

Le choix d'une bibliothèque nous permet de consacrer nos efforts aux seuls outils proposés, là où un langage devra être complet pour pouvoir être utilisé. Les autres outils et fonctions, de débogage, de compilation ou de développement graphique et mathématique par exemple, sont disponibles avec le langage support de la bibliothèque. Ces outils sont, de plus, optimisés, en constante évolution et adaptés à toutes les technologies présentes et à venir.

Notre simulateur étant un outil proposé à des informaticiens, la perte en convivialité, notamment pour la syntaxe, n'est pas un handicap, les informaticiens utilisant couramment des bibliothèques pour programmer. De plus, utilisant un langage courant, la syntaxe liée à ce langage est connue et maîtrisée par les utilisateurs potentiels. Ainsi l'investissement est minimal, nécessitant juste l'apprentissage des quelques fonctions proposées et du concept de programmation lié à la bibliothèque. De plus, la bibliothèque étant développée sur un langage courant, les utilisateurs ont la garantie de portabilité des implantations effectuées, même en cas d'abandon de la bibliothèque par son concepteur, le langage de base étant adapté à toutes les architectures et porté, avec des outils performants, sur les différents systèmes d'exploitations existants et à venir.

6.1.4 Le choix du langage : le langage C

Le choix du langage hôte de la bibliothèque était soumis à de nombreuses contraintes dues aux objectifs recherchés par le simulateur.

Le fait de permettre des implantations sur machines parallèles entraîne la recherche de performances et l'utilisation d'un langage performant en terme de rapidité des applications implantées à l'aide de ce langage. L'utilisation de machines parallèles implique l'utilisation d'un langage courant, disponible sur toutes les architectures actuelles et futures. Ce langage, en plus d'être présent sur tout type d'architectures MIMD doit, en outre, disposer de tous les outils permettant de programmer ces machines. Il doit pouvoir implanter les envois de messages et la mémoire partagée, c'est-à-dire supporter les bibliothèques MPI, PVM, Pthreads ou OpenMP. Ces seules contraintes limitent les possibilités aux seuls langages C et Fortran.

La contrainte de l'utilisateur, c'est-à-dire la contrainte d'utiliser un langage connu et utilisé par les utilisateurs de notre bibliothèque, les programmeurs de réseaux de neurones, limite le choix au langage C. C'est en effet le langage le plus utilisé par la communauté connexionniste, comme par toutes les communautés informatiques, avec le C++.

La recherche d'un langage permettant des applications efficaces rendait illusoire le choix du langage C++, celui-ci générant des applications plus lourdes que le C. De plus l'inclusion de primitives C dans un programme C++ étant triviale, les utilisateurs peuvent sans aucune difficulté utiliser une bibliothèque sur C dans des programmes écrits en C++. Enfin, le langage C++ n'est pas disponible immédiatement sur les nouvelles architectures parallèles.

6.2 Notre bibliothèque : une passerelle entre deux parallélismes distincts

Nous utilisons la technologie parallèle MIMD pour les exécutions parallèles et le parallélisme de neurones pour la description des modèles connexionnistes. Les propriétés du parallélisme de neurones vont servir de support à l'implantation sur le parallélisme matériel.

La bibliothèque devra gérer les différences de parallélismes des deux modèles, théorique et pratique. Ces différences portent essentiellement sur la granularité des parallélismes, les modes de

communication entre les unités et la synchronisation entre les unités parallèles des deux modèles. La bibliothèque se doit de masquer ces différences, c'est à dire de proposer aux utilisateurs d'utiliser le parallélisme des réseaux de neurones et de lui masquer les contraintes liées au parallélisme matériel.

6.2.1 Granularité

Nous proposons un simulateur de réseaux de neurones permettant à l'utilisateur de développer ses modèles en utilisant le parallélisme de neurones, c'est à dire les propriétés de calculs distribués des architectures connexionnistes, notamment des architectures d'inspiration biologique. Nous devons également garder à l'esprit que les modèles ainsi simulés devront être exécutés sur machine parallèle MIMD.

De ce fait, le simulateur doit proposer aux utilisateurs d'implanter des applications à *parallélisme à grain fin* et les exécuter, pour les exécutions sur machines parallèles, sous forme de *parallélisme à gros grain.*

Le grain fin du parallélisme neuronal

Pour générer des réseaux en utilisant le parallélisme de neurones, nous proposons de définir les réseaux au niveau des neurones, c'est à dire de définir, pour chacun des neurones du modèle, sa connectivité et la tâche qu'il doit accomplir pour déterminer son activité de sortie en fonction de l'information reçue en entrée. Nous obtenons ainsi un réseau d'unités, comprenant un grand nombre de connexions et supportant un fort trafic de communications. Même si les neurones utilisés peuvent être plus complexes que le neurone formel de McCulloch et Pitts, la tâche à effectuer par le neurone est de faible coût computationnel.

Nous obtenons ainsi un modèle de *parallélisme à grain fin*, chaque élément fonctionnant en parallèle et effectuant de courts calculs ponctués de nombreuses opérations de communications.

Le gros grain du parallélisme matériel

Nous avons vu dans la partie consacrée aux ordinateurs parallèles que les ordinateurs parallèles de types MIMD sont plutôt consacrés aux applications parallèles à gros grain, bien que permettant de simuler des applications à grains plus fin (voir section 4.2). La puissance des machines parallèles MIMD est essentiellement conséquence de la puissance des processeurs qui les composent, puisque ce sont des processeurs de technologie séquentielle classique. Si les communications entre ces processeurs sont performantes, l'obtention de performances sur ces ordinateurs nécessite des applications comprenant de longues phases de calcul, et un minimum de communications et de synchronisations entre les processeurs.

6.2.2 La communication

Comme pour la granularité, notre bibliothèque doit simuler un parallélisme par envoi de messages par l'intermédiaire de communications par partage de mémoire. Nos différents choix d'implantation font en effet que la sémantique d'implantation des réseaux connexionnistes entraîne une communication des neurones par **envoi de messages** tandis que les exécutions sur

machines parallèles, pour permettre des performances et les applications de réseaux connexionnistes de tout types, utiliseront des communications par **partage de mémoire** entre les différents processeurs de la machine.

La communication par envoi de messages des réseaux de neurones

Les réseaux de neurones artificiels, à l'image de leurs modèles naturels, sont des réseaux à grande connectivité et à communications fréquentes. Une condition préalable à la communication des neurones biologique est l'existence de connexions entre les neurones.

Pour un neurone biologique une connexion existe quand l'une des *dendrites* de l'une de ses *synapses* est fixée sur l'*axone* d'un neurone voisin, plus ou moins éloigné (voir section 1.1).

Une telle connexion permet au neurone de recevoir les impulsions électrochimiques émises par le neurone "connecté".

Ce protocole de communication s'apparente, au niveau informatique, à la communication par *envoi de messages* entre deux unités. La *décharge d'un neurone*, c'est-à-dire la transmission de son activation à travers sa sortie est équivalente à l'envoi d'un message du neurone à tous ses confrères préalablement connectés.

La communication par partage de mémoire sur machine parallèle

La programmation sur machines parallèles de type MIMD nécessite de gérer les communications entre les différents processeurs utilisés par les applications. Il est nécessaire, pour contrôler ces communications, de choisir entre les deux paradigmes actuellement disponibles sur les machines parallèles de type MIMD, la communication par envoi de messages et la communication par partage de la mémoire.

Le paradigme le plus immédiat serait l'envoi de messages : comme nous venons de le voir, les réseaux de neurones artificiels communiquent à l'aide de protocoles extrêmement proches de ce paradigme.

Il est malheureusement impossible d'implanter directement l'envoi de message des neurones sur machine parallèle.

Le choix de la technologie MIMD entraîne l'impossibilité de calquer le réseau de neurones artificiels sur le réseau de processeurs. La raison la plus évidente en est qu'une machine parallèle de ce type ne contient pas assez de processeurs pour affecter chaque neurone du réseau connexionniste à un processeur distinct. L'implantation impliquera une répartition des neurones sur un nombre réduit de processeurs.

L'utilisation d'envoi de messages *explicites* entre deux neurones pour simuler les communications entre ceux-ci entraîne une multiplication des messages échangés entre les processeurs. Les neurones ayant de très nombreuses entrées, chaque activation d'un neurone du réseau, qui implique un calcul faible, entraîne potentiellement **plusieurs messages échangés** du processeur hôte de ce neurone aux autres processeurs utilisés par l'application. Le nombre de communications entre les processeurs explose rapidement. A titre d'exemples, nous avons implanté, de cette façon, une carte auto-associative de Kohonen et un modèle appelé TOM [Durand, 1995], constitué de colonnes corticales comme unités de base. Ces implantations se sont faites à l'aide de ParCeL-1 et de MPI. Outre une extrême difficulté d'implantation, due aux envois et réceptions **explicites** de chaque message, les réseaux ainsi implantés voyaient leurs performances diminuer avec l'augmentation du nombre d'unités dans le réseau de neurones et du nombre de processeurs

utilisés. En effet, chaque ajout d'une nouvelle unité n'ajoutait que peu aux calculs effectués par le réseau, mais ajoutait un nombre considérable de communications.

Il est nécessaire de limiter le nombre de messages explicitement échangés entre les processeurs, et donc de regrouper les messages. Pour ce faire il faut regrouper les neurones à la manière de Reski [Reski, 1999], c'est à dire en plaçant sur le même processeur les neurones ayant les mêmes neurones connectés en entrées et en sortie, afin de n'avoir, idéalement, pour chaque processeur, qu'un processeur envoyant des messages et un seul en recevant. Ce regroupement ne peut être efficace que sur peu de modèles connexionnistes, principalement les réseaux à couches, les couches ayant un nombre de neurones équivalent ou proportionnel, et étant actives simultanément. De plus ce regroupement réclame une connaissance *a priori* de la topologie du réseau (quelles sont les sorties et les entrées de tous les neurones?) pour placer les différents neurones, ce qui interdit les modèles dynamiques.

Pour pallier ces difficultés, nous nous sommes intéressés à la communication par partage de mémoire entre les processeurs, paradigme plus adapté à la simulation des réseaux connexionnistes sur machines parallèles MIMD. Ce paradigme est plus souple, car il permet le partage des variables entre les processeurs de la machine. Il est présent actuellement sur toutes les nouvelles machines parallèle MIMD. De plus, les fabriquants investissant actuellement sur une norme commune, OpenMP, qui utilise des protocoles de partage de mémoire, nous pouvons penser que cette technologie va perdurer.

6.2.3 De l'asynchronisme biologique à l'implantation par cycles

Les réseaux de neurones biologiques sont des modèles non synchronisés, au sens informatique du terme. Néanmoins l'impossibilité technologique de simuler le mode de réaction d'un neurone à un stimulus, ajouté au déterminisme recherché par les applications simulées, entraîne la simulation de l'asynchronisme neuronal par un synchronisme par cycle des modèles développés.

L'asynchronisme théorique des modèles connexionnistes

L'asynchronisme théorique des réseaux de neurones artificiels est un concept hérité du modèle biologique. Dans le cerveau, le neurone réagit directement aux stimuli reçus par ses entrées : les impulsions électrochimiques reçues à travers ses synapses. Après avoir reçu une impulsion, la réaction du neurone est fonction de l'état de son environnement et de ses autres entrées. Il peut décharger son activité vers les autres neurones du cerveau, ou garder en mémoire la réception de l'activité afin de réagir à une stimulation future.

Ainsi les neurones ne sont pas synchronisés à une horloge centrale : ils réagissent en **temps continu** aux différents stimuli.

Dans le cas des réseaux de neurones artificiels, le principe se veut semblable. Les neurones déterminent leur sorties en fonction de leurs entrées, et ne déterminent donc leur activation qu'en fonction des activités reçues à travers leurs entrées. La grande différence réside dans le fonctionnement en **temps discret** des réseaux de neurones artificiels. Les neurones sont synchronisés par une horloge discrète. Ainsi un neurone artificiel détermine son activation, au temps $t+1$, en fonction de la valeur de ses entrées au temps t. Pour évaluer les neurones du réseau au temps t, il est indispensable d'avoir préalablement déterminé la valeur de tous les neurones au temps $t-1$ précédent, assurant ainsi le réseau de la disponibilité des valeurs d'entrées des différents

neurones (voir figure 6.1).

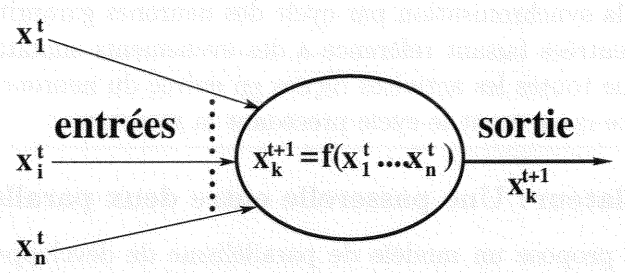


FIG. 6.1 – *Le synchronisme des modèles connexionnistes. Pour déterminer l'activation du neurone k au temps $t+1$, le réseau doit avoir déjà évalué les activations de ses neurones au temps t.*

Dans les modèles de réseaux de neurones statistiques, tous les neurones du réseau sont évalués à chaque pas de temps. Dans certains réseaux plus biologiques, seuls sont activés les neurones ayant reçu une impulsion en entrée. Certains réseaux, comme les modèles de Hopfield (voir section 2.2.5), sont dits *asynchrones*. Dans ces architectures connexionnistes les neurones sont activés selon des rythmes différents, et parfois irréguliers. Dans ce cas un neurone du réseau peut avoir été évalué à plusieurs reprises tandis qu'un autre ne l'a été qu'une seule fois.

La synchronisation par cycle de la bibliothèque

Notre bibliothèque doit permettre de simuler tout autant les modèles classiques que les modèles biologiques et, par là-même, les réseaux d'inspiration biologique asynchrones. L'utilisation du parallélisme de neurones signifie que tous les neurones d'un réseau ont la possibilité d'être actifs simultanément. Contrairement à la programmation traditionnelle, et séquentielle, il n'existe plus d'ordre d'évaluation des différents neurones présents dans le réseau.

Il peut sembler intéressant de se rapprocher du fonctionnement des neurones biologiques pour les exécutions et de revenir à des protocoles de mise à jour des neurones artificiels en temps continu. Mais l'utilisation de machines parallèles MIMD entraîne la gestion de plusieurs neurones pour chacun des processeurs, ce qui interdit, comme pour les exécutions séquentielles, une implantation en temps continu des neurones simulés. De plus, les algorithmes neuronaux actuels sont établis sur des protocoles de temps discrets, et nous souhaitons proposer aux utilisateurs la conservation de leurs algorithmes classiques.

Pour ces raisons, notre bibliothèque propose la simulation de neurones fonctionnant en **temps discret**.

Pour conserver le déterminisme des applications connexionnistes développées à l'aide de notre bibliothèque, nous proposons un mode de programmation **par cycles**. Tous les neurones d'un réseau exécuté seront synchronisés par un point de rendez-vous à la fin de chaque cycle d'exécution de l'application. Au cours de chacun des cycles, chaque neurone détermine, à partir de ses entrées, la valeur de son activité, qu'il transmet par l'intermédiaire de sa sortie. Le synchronisme par cycle nous permet d'utiliser le parallélisme de neurones sans perte d'informations et de déterminisme. En effet, ce synchronisme nous garantit la prise en compte, pour chacun des neurones du réseau, des informations reçues avant sa modification. Si un neurone du réseau était évalué deux fois entre deux modifications d'un autre neurone, connecté au premier, la première activité

du neurone source ne sera pas prise en compte par le réseau, ce qui modifie le résultat fourni par celui-ci. De même la synchronisation par cycle des neurones garantit la prise en compte, de la part du neurone, d'entrées faisant référence à des évènements simultanés, au sens du temps discret. Cela signifie que toutes les activités reçues en entrée du neurone auront été évaluées au cours du même cycle, ce cycle étant le cycle précédant la réception.

6.2.4 Notre simulateur : Une passerelle entre deux parallélisme distincts

Notre bibliothèque propose un modèle de parallélisme de développement très différent du parallélisme utilisé pour l'exécution sur machine parallèle.

Le parallélisme des réseaux de neurones artificiels L'utilisateur, c'est à dire le programmeur de réseaux connexionnistes utilisant notre simulateur, doit décrire ses réseaux comme un ensemble de petites unités à faibles capacités calculatoires, synchronisées par cycles, et communiquant par envoi de messages.

Le parallélisme de l'implantation parallèle L'implantation sur machine parallèle est une implantation à gros grain classique : les séquences de calcul sont regroupées et réparties sur les différents processeurs, les phases de communication et de synchronisation sont limitées et, afin de les optimiser, effectuées à l'aide de la communication par partage de mémoire.

Le rôle du simulateur, en plus d'être un outil de développement facilitant l'implantation des réseaux connexionnistes, est d'utiliser les propriétés du parallélisme des réseaux connexionnistes pour l'implantation sur machines parallèles. Nous cherchons à simuler un parallélisme à grain fin et envoi de message à l'aide d'un parallélisme à gros grain et mémoire partagée.

Chapitre 7

La description de notre simulateur

Nous décrivons dans cette partie du manuscrit le simulateur proposé. Nous souhaitons, par ce simulateur, apporter au monde connexionniste un outil lui permettant à la fois l'utilisation de machines parallèles et une convivialité accrue dans le développement des réseaux. Pour parvenir à ce résultat, notre simulateur doit non seulement masquer les spécificités des machines parallèles mais aussi être simple à utiliser pour des informaticiens connexionnistes mais non spécialistes en parallélisme.

Pour être attractif, notre simulateur propose un formalisme permettant de faciliter l'implantation des réseaux de neurones en rapprochant la programmation des modèles de la théorie du domaine.

7.1 Présentation de la bibliothèque

7.1.1 Objectifs du simulateur

Nous avons élaboré un simulateur généraliste et parallèle de réseaux connexionnistes. Il se présente sous la forme d'une bibliothèque de fonctions sur le langage C.

Ce simulateur s'adresse aux informaticiens du monde connexionniste. Il doit faciliter la manipulation de tous les types de réseaux de neurones artificiels, les réseaux mathématiques comme les réseaux plus directement inspirés de la biologie. Cette facilité de manipulation doit se traduire par une implantation plus simple et plus rapide mais aussi par un code induit plus intuitif et lisible et, par là-même, plus aisé à modifier.

Afin de pallier les difficultés dues à la taille des réseaux et aux temps de leurs exécutions, les programmes implantés doivent pouvoir être compilés et exécutés sur différentes architectures matérielles selon les besoins de l'utilisateur. Ainsi, nous proposons l'utilisation des plates-formes séquentielles classiques et des ordinateurs parallèles de type MIMD à mémoire partagée.

Cela permet, par exemple, de construire et corriger une architecture connexionniste sur machine séquentielle et d'effectuer les applications sur machine parallèle. Il est aussi envisageable d'effectuer l'apprentissage d'un réseau, coûteux en temps de calcul sur la machine parallèle et d'utiliser le réseau configuré sur plates-formes séquentielles.

Notre bibliothèque permet de simuler les réseaux de neurones à partir de leur unité de base, le plus souvent le neurone lui-même. L'utilisateur doit décrire la tâche affectée aux différentes unités de base de son modèle, charge est ensuite au simulateur d'exécuter les différentes unités pour faire émerger le comportement souhaité du réseau. Pour les implantations, nous proposons aux utilisateurs d'utiliser le parallélisme de neurones des réseaux connexionnistes (voir section 5.1),

c'est-à-dire de décrire les unités de ces modèles comme des unités autonomes, pouvant être évaluées sans ordre préétabli. La description des différentes unités du réseau doit décrire entièrement le réseau.

Les réseaux sont décrits comme **une somme d'unités autonomes, fonctionnant en parallèle et synchronisées par cycles.**

Pour la parallélisation, nous proposons d'utiliser les propriétés parallèles intrinsèques aux réseaux connexionnistes dans le développement pour profiter de la puissance des machines parallèles modernes.

Notre bibliothèque se veut donc un outil d'étude des réseaux de neurones artificiels dans le sens où elle permet d'user du caractère totalement distribué de ces réseaux dans leur étude et leur construction. Elle est aussi un outil d'application dans le sens où elle permet d'user de la puissance du parallélisme informatique pour exécuter les réseaux : cela permet d'accélérer les applications, de développer de plus larges modèles et donc de multiplier les propriétés de ceux-ci.

7.1.2 Les apports au connexionnisme

Notre simulateur se veut un outil plus riche qu'une bibliothèque de parallélisation des réseaux de neurones artificiels. L'expérience montre en effet qu'un tel outil n'est que peu utilisé par la communauté connexionniste. Ce type de simulateurs demande une adaptation des algorithmes des modèles et une nouvelle implantation de ceux-ci de la part des programmeurs.

Nous souhaitons, pour notre part, proposer un outil apportant une aide au développement des réseaux de neurones et permettant d'utiliser une même implantation de réseau sur des machines cibles distinctes sans **aucune modification, ni d'algorithme, ni d'implantation**. Les algorithmes de base des réseaux de neurones doivent pouvoir être utilisés et un même code exécuté sur machine séquentielle classique ou sur machine parallèle MIMD.

Une utilisation d'un formalisme proche du modèle théorique

Un réseau de neurone est habituellement décrit comme une somme de neurones connectés, neurones dont les connexions et les actions sur ces connexions sont décrites. Les propriétés des réseaux émergent ensuite de la somme des actions locales des différents neurones du réseau.

De la même manière, notre simulateur décrit un réseau connexionniste comme une somme de neurones fonctionnant en parallèle. L'implantation d'un modèle connexionniste se fait donc au niveau des neurones de ce modèle.

Une vision locale et distribuée des exécutions

Les implantations classiques des réseaux connexionnistes se font habituellement en simulant les différentes variables locales des neurones à l'aide de différents tableaux de variables globales de l'exécution (tableaux des poids, tableaux des sorties, tableaux des variables locales, etc.). L'exécution proprement dite pioche dans les différents tableaux les valeurs correspondant au neurone évalué. De plus, l'ordre d'évaluation des différents neurones du réseau est totalement **déterminé** par le programmeur. La gestion des variables globales du programme est donc contrôlée par le programmeur et le caractère parallèle de l'évaluation des neurones au sein d'un réseau est totalement perdu.

De la même manière, toutes les variables locales d'un neurone sont potentiellement accessibles à tout moment de l'exécution, et donc pour toute opération concernant un autre neurone du réseau. Les connexions énoncées dans la théorie des réseaux de neurones artificiels ne sont

pas explicitées dans les implantations (voir figure 7.1). Le programmeur ne prend pas en compte l'acheminement et la disponibilité *théorique* des variables qu'il utilise pour un calcul de neurone, calcul qui devrait être totalement local et exécuté à partir des seules variables accessibles au neurone.

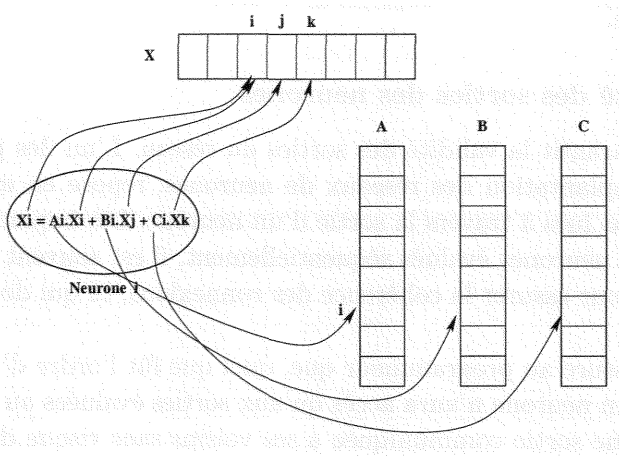


FIG. 7.1 – *Gestion des données dans une implantation de réseaux de neurones. Toutes les données sont globales.*

Avec notre simulateur, chaque neurone du réseau est un objet indépendant, qui possède ses propres variables locales (voir figure 7.2). L'ordre d'évaluation des différents neurones du réseau n'est ainsi plus à la charge du programmeur et les neurones peuvent potentiellement être exécutés dans n'importe quel ordre. La bibliothèque garantit seulement que tous les neurones auront été évalués pour le cycle t avant l'évaluation du premier neurone au cours du cycle $t + 1$.

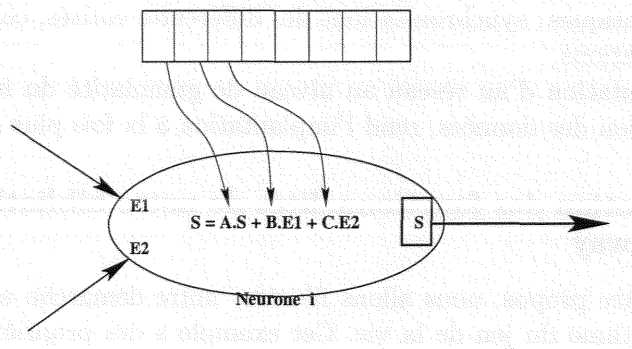


FIG. 7.2 – *Nous proposons une implantation dans laquelle chaque neurone dispose, comme variables, de ses données propres, de ses entrées et de sa sortie*

Un accès aux machines parallèles

L'une des principales faiblesses des réseaux de neurones est due au coût, en temps, de ces modèles. La multiplication du nombre de neurones, pour augmenter la puissance des réseaux, pose des problèmes de temps d'utilisation, et surtout d'initialisation pour les modèles supervisés. L'utilisation de réseaux de neurones artificiels dans l'étude des comportements corticaux entraîne

aussi une inflation dans la taille des réseaux implantés, et un besoin toujours plus conséquent en puissance de calcul.

La possibilité d'exécuter un réseau utilisant notre bibliothèque sur machines parallèles répond aussi à ce besoin et permet d'envisager de simuler des réseaux de plus grande taille. Cette potentialité peut notamment être intéressante dans le cadre de la recherche sur les réseaux d'inspiration biologique.

Gestion de la validité des sorties des neurones

Notre simulateur garantit la validité des sorties du réseau. L'un des problèmes fréquemment rencontrés, lors de l'implantation des réseaux de neurones, repose en effet sur ce problème de validité des informations lues à travers la sortie d'un neurone. Les implantations étant effectuées en temps continu et les neurones évalués séquentiellement, il est souvent nécessaire de recourir à des variables *tampon* pour assurer la cohérence des connexions, ce qui double les variables gérées par l'utilisateur.

Notre simulateur assure au programmeur que, quel que fût l'ordre d'évaluation des neurones du réseau, au temps t un neurone n'aura accès qu'aux sorties évaluées au temps $t-1$. Le neurone peut ainsi manipuler une sortie communiquée à ses voisins sans risque d'incohérences.

Aide au débogage

Pour faciliter le développement des modèles connexionnistes, nous proposons une aide au débogage. Notre simulateur permet, en phase de développement des réseaux, de vérifier certains de leurs aspects topologiques et d'aider ainsi le programmeur à développer et corriger ses implantations.

Notre bibliothèque propose des fonctions qui permettent aux utilisateurs de construire différentes entités et de les connecter entre elles. Nous gérons les difficultés d'implantation consécutives à ces choix sémantiques : synchronisations des différentes entités, cohérence des variables et gestion des communications.

Pour finir, l'implantation d'un réseau au niveau de granularité du neurone, sans se soucier des problèmes de gestion des données, rend l'implantation à la fois plus simple et plus lisible.

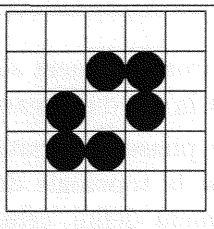
7.1.3 Une illustration des fonctionnalités de notre bibliothèque : *Le jeu de la vie* de Conway

Pour expliciter notre propos, nous allons illustrer notre démarche avec des exemples d'implantation d'un algorithme du jeu de la vie. Cet exemple a des propriétés proches des réseaux de neurones en termes de topologie et de paradigmes calculatoires. Il ne nécessite, en revanche, aucun apprentissage. Notre propos ne visant pas à étudier les algorithmes d'apprentissage mais à élaborer des outils de construction des réseaux, le jeu de la vie convient bien à notre démarche. Il est possible de le voir comme un réseau sans apprentissage ni entrées.

Ce jeu est composé d'une carte d'automates cellulaires ou cellules. Chaque cellule peut adopter deux états: vivante ou morte, états représentés par une valeur d'activation de 1 ou de 0. Une relation de voisinage immédiat est à l'origine de deux règles:

Si une cellule possède 3 voisines vivantes (procréation)
ALORS elle devient vivante (activité = 1).

SI une cellule a plus de 3 voisines vivantes (étouffement)
OU
SI elle a moins de 2 voisines vivantes (isolement)
ALORS elle meurt (activité = 0).



Ce formalisme simple est assez proche, du point de vue fonctionnel et topologique, des réseaux neuromimétiques. Il va nous permettre d'illustrer d'exemples les différentes phases de construction d'un réseau de neurones artificiels.

Nous montrerons, tout au long de la présentation de notre bibliothèque, comment se construit et s'implante un jeu de la vie avec notre formalisme et les fonctions de notre simulateur.

7.2 Utilisation de la bibliothèque pour construire des réseaux connexionnistes

7.2.1 Les différentes étapes d'exécution d'un réseau

L'exécution d'un réseau implanté à l'aide de notre simulateur se fait en plusieurs étapes.

La première étape correspond à la définition des caractéristiques de départ de l'exécution. Il est nécessaire de définir le réseau qui commencera l'exécution et le nombre de processeurs utilisés pour cette exécution en cas d'utilisation de machine parallèle. Le réseau de départ se définit en spécifiant les neurones présents lors de la première itération du réseau (voir figure 7.3). A ce moment, aucune topologie n'est encore spécifiée, la charge de la construction de l'architecture connexionniste étant distribuée sur les différents neurones.

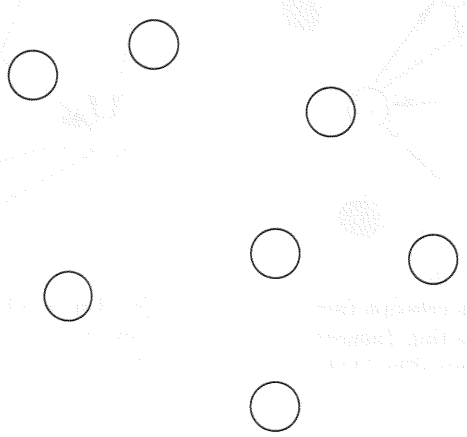


FIG. 7.3 – *Avant de démarrer l'exécution du réseau l'utilisateur définit les neurones présents au départ de l'exécution.*

La seconde étape correspond à l'exécution proprement dite du réseau, exécution rythmée par les cycles d'exécution des neurones. Lors de cette étape, le réseau devient autonome. En

effet seuls les neurones influent sur l'exécution, le programmeur ne dispose d'aucun autre moyen d'intervenir dans la vie du réseau.

Durant cette phase d'exécution du réseau les neurones créent chacun leurs connexions et déterminent ainsi la topologie du réseau (voir figure 7.4). Si cette exécution démarre avec un réseau préalablement défini, celui-ci peut évoluer à l'envie en créant et éliminant des neurones (voir figure 7.5).

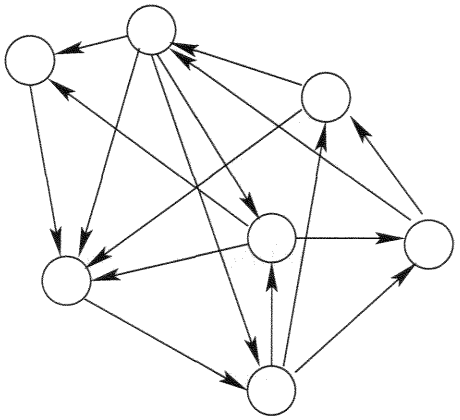
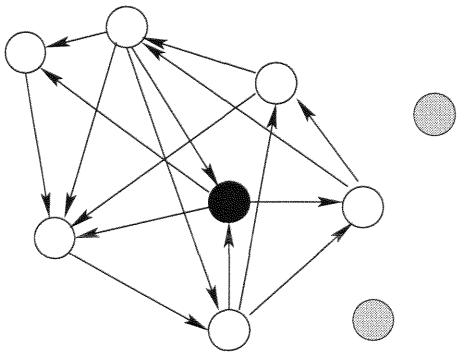
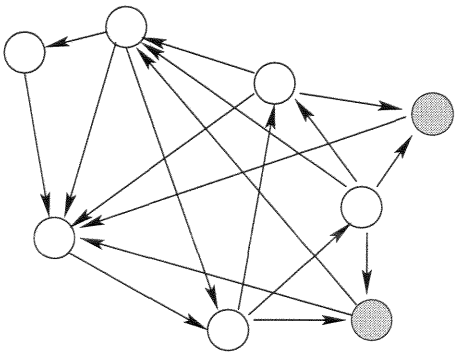


FIG. 7.4 – *Au cours de l'exécution, la topologie du réseau est définie par la création des différentes connexions de chacun des neurones.*



(a) Le réseau demande la création (neurones gris) et la destruction (neurone noir) de neurones au cours d'un cycle



(b) Les modifications sont prises en compte, ce qui modifie la topologie du réseau.

FIG. 7.5 – *Au cours de l'exécution du réseau la topologie du réseau est dynamique. Ainsi lors d'un cycle, certains neurones peuvent être éliminés, tandis que d'autres sont créés, voir figure 7.5(a). La topologie du réseau est donc modifiée lors des cycles suivants, voir figure 7.5(b).*

A la mort de tous les neurones du réseau, l'exécution se termine. Il n'existe pas d'alternative pour terminer l'exécution d'un réseau.

7.2.2 La construction d'un réseau connexionniste

Pour construire un réseau connexionniste, un utilisateur devra construire et implanter son réseau sous forme de neurones **autonomes** et pouvant évoluer en **parallèle**. Les neurones ne communiquent que par l'intermédiaire de leurs **connexions**.

Les neurones peuvent en effet être exécutés dans n'importe quel ordre, et, en cas d'exécution sur machine parallèle, simultanément sur différents processeurs (voir figure 7.6). Ils ne connaissent d'un autre neurone que sa sortie, à condition de s'y être préalablement connectés.

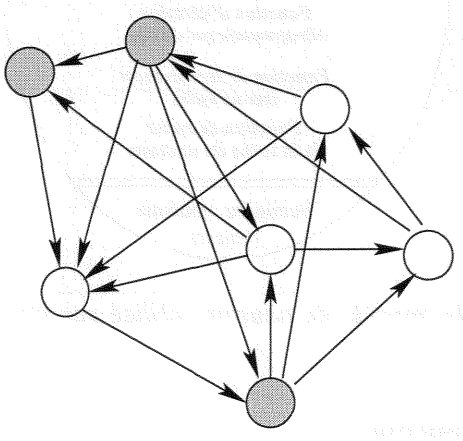


FIG. 7.6 – *Les neurones d'un réseau doivent pouvoir fonctionner en parallèle. Cela signifie que deux neurones peuvent être évalués simultanément au cours d'un même cycle. Par exemple, les neurones représentés en gris sont évalués en même temps.*

Pour construire un réseau connexionniste à l'aide de notre bibliothèque, le programmeur devra donc définir explicitement chacun des neurones de son réseau et la tâche assignée à ceux-ci lors de chacun de leurs cycles de vie. En fait, l'utilisateur devra définir explicitement les différents types de neurones présents dans son réseau et déclarer chacun des neurones du réseau en spécifiant son type.

7.2.3 Définir un neurone

La bibliothèque définit un neurone[28] comme un objet possédant ses variables locales, une sortie unique, et un certain nombre de connexions (ses entrées) avec d'autres neurones du réseau (voir figure 7.7). La bibliothèque divise le temps d'exécution d'un réseau de neurones en cycles. A chaque cycle, chaque neurone réévalue sa sortie en fonction de ses entrées et de ses variables locales.

Définir un neurone de notre simulateur consiste à spécifier les variables locales, la sortie, la fonction de copie de cette sortie et les connexions du neurone. Il est aussi nécessaire d'implanter le code décrivant la tâche à effectuer au cours de chaque cycle. Chaque neurone d'un réseau possédera, pour l'identifier, une immatriculation.

28. Neurone est ici utilisé comme terme générique de l'unité de base du réseau implanté. Selon les besoins de l'utilisateur, cette unité de base peut devenir, par exemple, une colonne corticale ou une maxi-colonne.

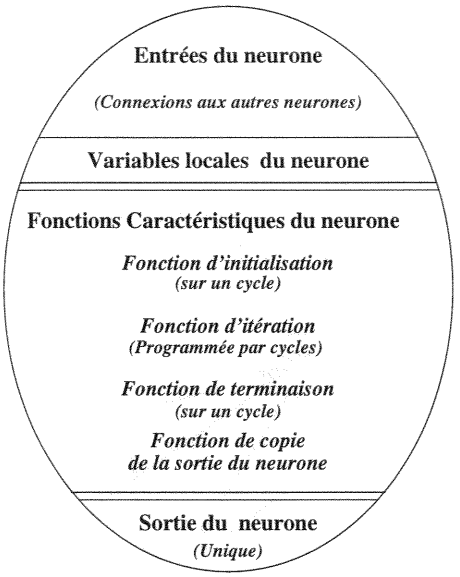


FIG. 7.7 – *Le modèle de neurone utilisé par notre simulateur*

L'immatriculation d'un neurone

Dans notre bibliothèque, un neurone est identifié par son numéro d'immatriculation. Cette immatriculation, unique, permet aux neurones de se connecter entre eux. Cette immatriculation étant une variable du neurone, elle peut lui permettre de s'identifier lui-même afin, par exemple, de spécifier ses connexions. Le choix du numéro d'immatriculation du neurone est laissé à l'utilisateur.

Nous verrons dans la figure 7.8 comment, à partir d'immatriculations judicieusement choisies, une cellule du jeu de la vie peut déduire les immatriculations de ses cellules voisines de sa propre immatriculation. Ces immatriculations connues, elle peut ensuite construire ses connexions.

L'immatriculation d'un neurone lui est attribuée lors de la demande de création de ce neurone.

Les fonctions caractéristiques du neurone

Le corps du neurone est implanté sous forme de fonctions. Ces fonctions contiennent la tâche impartie au neurone, ainsi que la définition de ses différentes variables. Elles s'implantent en langage C enrichi des fonctions fournies par la bibliothèque.
Pour simplifier l'implantation et sa lisibilité, le corps d'un neurone se divise en trois fonctions que nous appellerons **fonctions caractéristiques du neurone** : initialisation, itération et terminaison.

La fonction d'initialisation Elle décrit les tâches à accomplir par le neurone lors de sa création. Elle sert le plus souvent à définir et allouer les différentes variables du neurone, à spécifier sa sortie et à créer ses connexions. Elle est exécutée une unique fois, au cours du premier cycle de vie du neurone.

La fonction d'itération Cette fonction décrit la méthode utilisée par le neurone pour déterminer la valeur de sa sortie et les différentes tâches qu'il doit exécuter au cours d'un cycle de vie.

Il est par exemple possible de créer ou de tuer des neurones ou des connexions. Elle est exécutée du second au pénultième cycle de vie du neurone, jusqu'à ce que le neurone soit tué.

La fonction de terminaison Elle correspond à la tâche à accomplir lors du dernier cycle de vie du neurone, au cycle suivant celui où la mort du neurone fut demandée. Elle sert essentiellement à désallouer les différentes variables du neurone.

Les entrées du neurone

Elles correspondent aux liens reliant le neurone à d'autres neurones du même réseau. Ces liens sont des *canaux de connexion feed-forward.* Ils permettent un accès à la sortie du neurone connecté. Ainsi, une fois le neurone connecté, les entrées peuvent être vues comme des variables locales du neurone, manipulables uniquement en lecture.
Les connexions ne permettent la communication que du neurone connecté au neurone "connecteur". Ainsi ce dernier ne peut que recevoir de l'information par ce canal (voir section 7.2.5), il ne peut en envoyer au neurone connecté que par l'intermédiaire d'une éventuelle seconde connexion, dite *connexion retour.*

Les variables locales du neurone

Un neurone peut posséder des variables locales. Ces variables locales sont les variables utilisées d'un cycle à l'autre par le neurone qui lui sont personnelles. Elles représentent les *variables privées* du neurone.
En termes de programmation, ces variables sont représentées par **un pointeur**, que le neurone utilise à chaque cycle, et que la bibliothèque se charge de lui fournir. Cette adresse pointe sur un élément de type quelconque, simple ou structuré, selon les choix et les besoins de l'utilisateur. Ce pointeur permet de ne manipuler qu'un élément, ce pointeur, tout en laissant la liberté à l'utilisateur d'y placer le nombre d'éléments désiré. Les variables locales du neurone sont donc implantées sous forme d'une unique variable locale, représentée par son pointeur. Ce pointeur est généralement affecté dans la fonction d'initialisation du neurone, récupéré dans la fonction d'itération et désalloué dans la fonction de terminaison.
Cette variable locale permet au neurone de fonctionner sur ses variables propres. Ainsi tout neurone peut demander à utiliser une variable α, qui aura une valeur spécifique pour chacun des neurones. Dans un programme séquentiel classique, l'utilisateur devrait gérer un tableau de α, contenant une valeur spécifique pour chacun des neurones.
Généralement cette variable locale contient, au moins, les identificateurs de canaux de connexion des neurones, mais elle peut contenir toute autre variable selon la définition de l'utilisateur, les variables d'état du neurone par exemple.

La sortie du neurone

A l'image de son grand frère biologique, le neurone artificiel de notre simulateur n'admet qu'une unique sortie. Cette sortie est le seul lien dont dispose le neurone pour communiquer avec le monde extérieur, c'est-à-dire avec les autres neurones du réseau.
Définir la sortie du neurone correspond, pour le programmeur, à spécifier une variable. Cette variable sera accessible en lecture aux neurones du réseau qui se seront préalablement connectés. Comme pour les variables locales, le type de la sortie d'un neurone n'est pas imposé, il est laissé à l'appréciation de l'utilisateur. Cette tolérance permet, principalement, de simuler des réseaux reposant sur une granularité autre que le neurone. Il est ainsi possible de travailler au niveau de la

colonne corticale ou de la maxi-colonne, unités qui possèdent plusieurs sorties. La liberté de type permet aussi de simuler des réseaux dont la topologie se marie difficilement à notre approche. Il est ainsi possible de simuler un perceptron multi-couches et sa rétro-propagation du gradient avec nos connexions uniquement feed-forward.

Cette sortie est spécifiée à la librairie sous forme de pointeur sur la variable choisie, accompagnée de sa **fonction de copie de sortie**. Généralement la déclaration de la sortie du neurone s'effectue au sein de sa *fonction d'initialisation.*
C'est cette sortie que les neurones préalablement connectés pourront lire. Ils auront ainsi accès à la valeur de la sortie déterminée au cours du cycle précédant la lecture. En effet le neurone peut manipuler et modifier sa variable de sortie sans risque de modifier la valeur lue de l'extérieur, ce qui garantit la cohérence des connexions.

La fonction de copie de sortie du neurone

Cette fonction doit accompagner toute déclaration d'une sortie pour un neurone. Elle définit une méthode permettant d'effectuer une copie exacte de la sortie du neurone à partir du pointeur de celle-ci.
Cette fonction est essentielle à la bibliothèque. Elle lui permet de garantir la cohérence des données lues à travers les connexions des neurones connectés et de laisser libre le choix du type de la sortie.

Les variables globales du réseau

Tout neurone du réseau a aussi un accès aux variables globales du programme. Cette spécificité n'est pas particulière à notre bibliothèque, elle correspond aux propriétés habituelles des fonctions implantées en C. Le programmeur peut ainsi utiliser les variables globales du réseau pour implanter les données auxquelles plusieurs neurones de son réseau accèdent.

Les entrées des neurones, généralement implantées sous forme de tableaux ou de fichiers, peuvent ainsi être placées dans les variables globales du réseau.

L'exemple du jeu de la vie

Comme nous l'avons présenté ci-dessus, pour définir la grille de cellules du jeu de la vie il est nécessaire de définir chaque cellule de cette grille. Il faut donc définir, pour chaque cellule du jeu, les variables locales, la sortie, la fonction de copie de cette sortie et les connexions de celle-ci.

L'immatriculation Elle est affectée à la cellule lors de sa création. Pour faciliter la lecture et la programmation des cellules il est souhaitable d'attribuer des immatriculations *signifiantes* aux cellules, comme par exemple les immatriculations présentées dans la figure 7.8. Cette numérotation permet aussi de facilement déterminer les immatriculations de connexions invalides pour un neurone. En effet les cellules des bords ne peuvent pas se connecter à huit voisins, certains n'existant pas. Ce problème des cellules de bord de grille n'apportant aucune information sur notre simulateur, il ne sera plus évoqué par la suite.

La variable locale de la cellule Elle comprendra les huit canaux de connexion de la cellule et son activité. Il faudra donc déclarer une variable d'un type structuré comme celle présentée dans l'exemple 1.

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	*32*	*33*	*34*	35	36	37
40	41	*42*	**43**	*44*	45	46	47
50	51	*52*	*53*	*54*	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

FIG. 7.8 – *Affectation possible des immatriculations pour une grille de jeu de la vie (8×8). Cette affectation permet au neurone, à partir de son immatriculation, de connaître rapidement les immatriculations de ses huit voisins directs.*

```
typedef struct {
  canal_in    *liens;       /* Tableau de canaux de connexions */
  int         activite;     /*     Activité de la cellule      */
} typevariable;
```

Exemple 1 : *Type des variables locales des cellules*

La fonction d'initialisation La fonction d'initialisation permet à chaque cellule du jeu d'allouer un pointeur sur ses variables locales et de définir sa sortie, c'est à dire la valeur de son activité, un entier. Elle permet de plus à la cellule d'allouer son tableau de connexions et de les créer. Les cellules auxquelles se connecter se retrouvent rapidement avec la méthode montrée dans la figure 7.9. Chaque cellule se connecte ainsi à ses huit cellules voisines (voir exemple 3).

-11	*-10*	*-9*
-1	**immat**	*+1*
+9	*+10*	*+11*

FIG. 7.9 – *Toute cellule peut retrouver rapidement les immatriculations de ses huit voisines, à partir de sa propre immatriculation* ***immat***

La fonction d'itération La fonction d'itération permet de définir, lors de chaque cycle du jeu, la valeur de l'activation de la cellule. Pour ce faire elle additionne les valeurs reçues par ses entrées, valeur correspondant aux activités des cellules voisines obtenues lors du cycle précédent. Selon le résultat de cette addition, la cellule détermine sa propre activation (voir exemple 4).

La fonction de terminaison Elle servira, lors du dernier cycle de vie de la cellule, à désallouer le tableau de connexions de la cellule (voir exemple 5).

La sortie de la cellule Il s'agit ici de la valeur intéressant les autres cellules connectées. Dans notre cas, la sortie sera l'activité de la cellule (0 ou 1) implantée sous forme d'un entier (voir exemple 3).

La fonction de copie de sortie Elle définit une méthode permettant d'obtenir un pointeur sur un entier de même valeur que la sortie de la cellule (voir exemple 2).

```
int *CopieSortieCellule(int *activite)
{
 int *tampon;

 tampon = (int *) malloc(sizeof(int));

 *tampon = *activite;

 return(tampon);
}
```

Exemple 2 : *Fonction de copie de la sortie d'une cellule pour le jeu de la vie*

```
void InitCellule()
{
 typevariable *VarLocale;

 VarLocale = (typevariable *) malloc(sizeof(typevariable));

 VarLocale->liens = (canal_in *) malloc(8*sizeof(canal_in));

 /* Déclarer (*VarLocale) comme variable de la cellule      */
 /* Déclarer &(VarLocale->activite) comme sortie du neurone */
 /* Connexion aux huit cellules voisine                     */
}
```

Exemple 3 : *Squelette de la fonction d'initialisation d'une cellule du jeu de la vie*

7.2.4 Création et destruction des neurones

Il existe deux méthodes distinctes de création des neurones, dites *statique* et *dynamique*.
La méthode statique permet de définir les neurones présents à la création du réseau. Ces neurones sont créés dans le programme principal du réseau.
La méthode dynamique permet de définir de nouvelles unités au cours de l'exécution du réseau et de développer ainsi des architectures connexionnistes évolutives. Ces neurones sont créés par des neurones présents dans le réseau, au cours de leur exécution. Les nouveaux neurones seront effectivement créés, leur fonction d'initialisation sera exécutée, au cycle suivant la demande de création (voir figure 7.5).

La destruction des neurones se fait en trois étapes réparties sur trois cycles consécutifs (voir figure 7.10).

- La demande de la mort du neurone (figure 7.10(a)).

```
void IterCellule()
{
 /* Récupérer la variable locale de la cellule              */
 /* Effectuer la somme des variables reçues par les entrées */
 /* Evaluer sa sortie                                       */
 si Somme = 3 alors VarLocale->activité = 1;
              sinon VarLocale->activité = 0;
}
```

Exemple 4 : *Squelette de la fonction d'itération d'une cellule du jeu de la vie*

```
void TermCellule()
{
 /* Récupérer la variable locale de la cellule              */

 free(VarLocale->liens);
 free(VarLocale);
}
```

Exemple 5 : *Squelette de la fonction de terminaison d'une cellule du jeu de la vie*

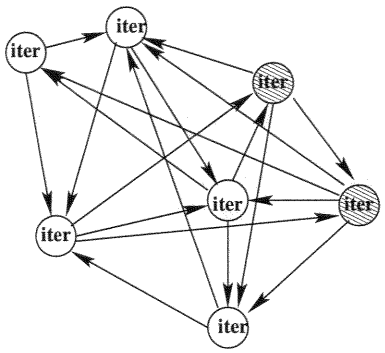
- L'exécution de la fonction de terminaison du neurone (figure 7.10(b)).
- La disparition effective du neurone (figure 7.10(c)).

La demande de destruction peut être effectuée par le neurone lui-même ou par tout autre neurone du réseau. A la mort d'un neurone ses connexions sont détruites et les connexions sur celui-ci pointent sur **NULL**.

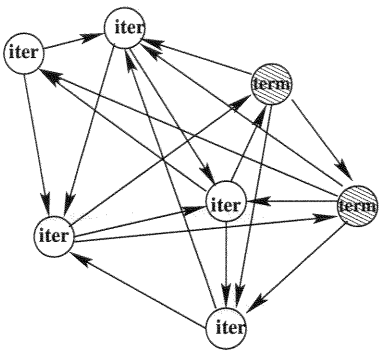
7.2.5 Les connexions

En utilisant notre bibliothèque, la topologie d'un réseau neuromimétique est définie par l'ensemble des connexions de chacun des neurones qui le composent. Les connexions entre les différentes entités sont commandées localement par chacun des neurones. Les connexions sont des canaux construits entre la sortie d'un neurone et l'entrée d'un autre neurone. Ces connexions sont de type *feed-forward* et elles assurent la cohérence des données transmises au cours d'un cycle.

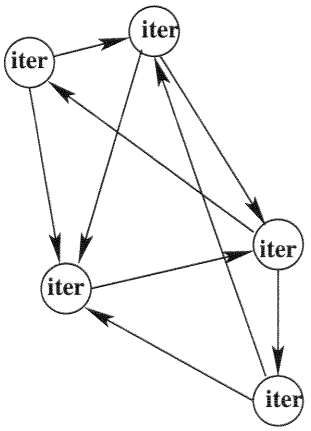
Une connexion détermine un canal de communication Dans notre modèle (voir figure 7.7), un neurone dispose d'une unique sortie, qui est à la disposition de son environnement. Il est à noter que le type -au sens informatique du terme- de cette sortie est totalement libre, il peut même être structuré. Cette sortie est la seule variable d'un neurone accessible aux autres neurones du réseau. Pour avoir accès à cette variable un neurone doit se connecter au neurone propriétaire, créant ainsi un canal de communication, canal à travers lequel il pourra ensuite lire la sortie du neurone auquel il est connecté (voir figure 7.11). Les canaux ainsi définis constituent tout à la fois les entrées des neurones et la topologie du réseau.



(a) Cycle t. Le réseau a demandé la mort des neurones grisés.



(b) Cycle $t+1$. Les neurones grisés exécutent leur fonction de terminaison.



(c) Cycle $t+2$. Les neurones grisés ont effectivement disparus.

FIG. 7.10 – *La mort d'un neurone se fait en trois étapes sur trois cycles de vie du réseau.*

La création d'une connexion peut se faire sur tout neurone du réseau. Il n'est pas nécessaire que celui-ci ait déclaré sa sortie. En revanche, il est impossible de lire à travers une connexion avec un neurone n'ayant pas défini sa sortie.

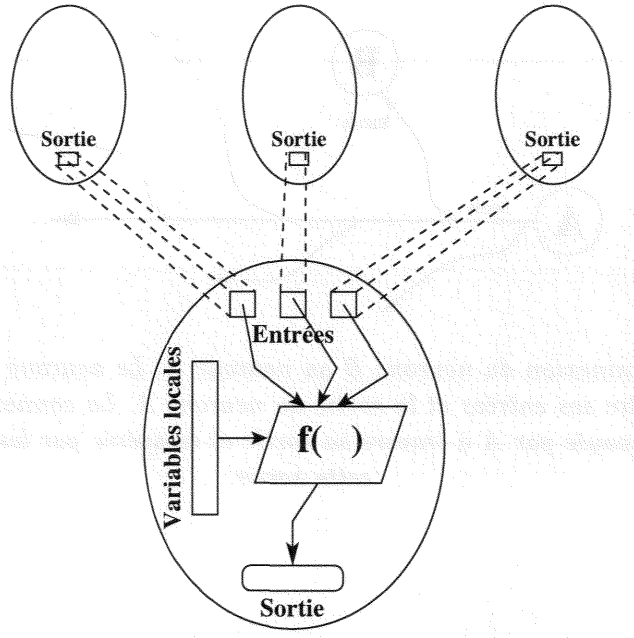


FIG. 7.11 – *La gestion des connexions vue par le neurone. Les entrées permettent au neurone d'utiliser les sorties des neurones comme des variables propres. Les canaux de communication lui donnent en effet un accès direct, en lecture, aux variables de sortie des neurones connectés.*

Des connexions *feed-forward* Les connexions proposées par notre bibliothèque sont de type *feed-forward*. Seul le neurone connecté peut utiliser le canal de connexion créé. Lorsqu'un neurone B se connecte à un neurone A, les informations transmises par le canal vont de **A vers B**, il est impossible de transmettre de l'information de B vers A par ce canal (voir figure 7.12). De plus seul le neurone B est informé de la connexion[29]. Un neurone définit sa sortie mais il ne dispose d'aucune information sur le nombre ou l'identité des neurones connectés à celle-ci.

La cohérence des données Notre bibliothèque garantit à l'utilisateur que, quel que soit l'ordre d'évaluation des neurones au sein du réseau, au cours d'un cycle t, un neurone lira toujours la valeur de la sortie des neurones évaluée au cours du cycle précédent, voir figure 7.13. En d'autres termes, la cohérence assure que tous les neurones connectés à un neurone H recevront une valeur de sortie de H identique au cours d'un même cycle. Cette propriété évite la gestion de la cohérence des entrées au programmeur, gestion généralement effectuée à l'aide de *variables tampon* [Rougier, 2000].

Pour assurer cette cohérence des variables de sortie des neurones, une **fonction de copie de la sortie du neurone** est demandée pour chaque neurone ayant déclaré une sortie. Cette fonction définit une méthode permettant de créer un pointeur sur une variable **totalement équivalente**

29. Il peut demander à tout moment l'immatriculation du neurone origine d'un canal de communication.

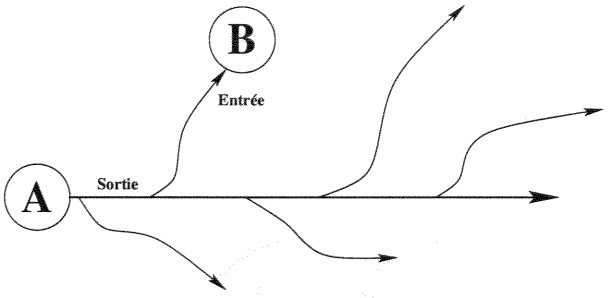


FIG. 7.12 – *Connexion du neurone B au neurone A. Le neurone crée un canal de communication entre ses entrées et la sortie du neurone A. La connexion est feed-forward, l'information est envoyée par A à travers sa sortie et récupérée par les neurones connectés à cette sortie.*

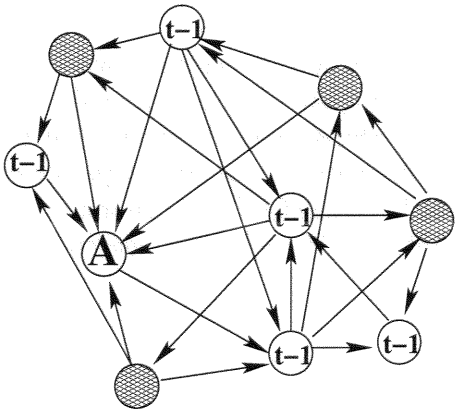


FIG. 7.13 – *La bibliothèque assure la cohérence des données reçues par les entrées des neurones. Lorsque le neurone A est évalué lors du cycle t alors que certains neurones, grisés, ont déjà été évalués pour ce cycle et d'autres non (les neurones en blanc) la bibliothèque garantit que les entrées reçues par le neurone A auront été calculées au cours du même cycle, le cycle* $t-1$.

à la sortie du neurone qui lui est associée. Cette fonction est donnée à la bibliothèque qui se charge de son appel et donc des allocations exécutées par cette fonction. A la mort d'un neurone la bibliothèque se charge de désallouer les variables déclarées lors de l'appel de cette fonction.

Les topologies peuvent être dynamiques Notre simulateur propose la construction de tout type de réseaux, réseaux évolutifs compris. Ainsi tout neurone du réseau peut, au cours de chacun de ses cycles de vie, initialiser de nouveaux canaux de communication ou détruire certains de ses canaux. Les créations de connexion peuvent être implantées dans la fonction d'initialisation comme dans la fonction d'itération d'un neurone. Il faut signaler que la mort d'un neurone implique la destruction de ses canaux de communication, l'utilisateur n'a donc pas besoin d'éliminer les canaux au cours de la fonction de terminaison. Par contre les connexions sur des neurones sur la sortie du neurone ne sont pas détruites, les neurones connectés recevront par leurs canaux un pointeur nul.

Une communication en deux phases La construction d'une communication entre deux neurones se fait en deux étapes, il est en effet nécessaire de construire le canal de communication avant de pouvoir y lire une information. La connexion se fait sur deux cycles de vie d'un neurone, un canal de communication est ainsi utilisable au cycle suivant la demande de connexion de la part d'un neurone.

7.3 Les fonctions offertes par la bibliothèque

Pour implanter des réseaux selon la méthodologie décrite ci-dessus, nous proposons une bibliothèque sur le langage C. Pour faciliter l'utilisation de cette bibliothèque, elle ne comprend qu'un nombre assez limité de fonctions et macros, permettant de décrire les réseaux connexionnistes, les neurones et les connexions. Le reste de l'implantation peut s'effectuer à l'aide des différentes fonctions et des différents outils proposés par le langage C.

Nous proposons maintenant une brève description des fonctions de notre simulateur. Une description plus technique est fournie en annexe de ce manuscrit. Les macros proposées le sont essentiellement pour alléger la syntaxe inhérente à l'écriture des réseaux. Nous présenterons en annexe les différences d'implantation apportées par ces macros.

7.3.1 La définition et l'exécution du réseau de départ

Avant de démarrer l'exécution d'un réseau de neurones, il est nécessaire d'indiquer à la bibliothèque la configuration de départ de ce réseau. Cette configuration se fait en indiquant le nombre de neurones présents au début de l'exécution du réseau, puis en explicitant chacun de ces neurones à l'aide de leurs fonctions caractéristiques respectives. Il est aussi nécessaire de préciser le nombre de processeurs souhaité en cas d'utilisation d'un ordinateur parallèle. Ce nombre de processeurs n'est pas pris en compte en cas d'exécution du réseau sur une plate-forme séquentielle.

Pour communiquer à la bibliothèque la définition du réseau de départ et lancer l'exécution de ce réseau, l'utilisateur doit obligatoirement utiliser les trois fonctions suivantes dans le corps de son programme.

InitNetwork Cette fonction permet de spécifier à la bibliothèque le nombre de neurones présents au premier cycle d'exécution du réseau, et le nombre de processeurs utilisés pour l'exécution

de celui-ci, en cas d'utilisation d'une machine parallèle. Cet appel permet à la bibliothèque de spécifier les différentes variables nécessaires à l'exécution du réseau et, en cas d'exécution sur machine parallèle, les fonctions et variables permettant l'utilisation de ces machines.

ExecuteNetwork Cette fonction permet de lancer l'exécution du réseau. Cette exécution se fait avec les neurones préalablement définis. Après cet appel, le réseau est exécuté jusqu'à ce qu'il ne contienne plus de neurones.

FreeNetwork Cette fonction permet à la bibliothèque de désallouer toutes les variables internes issues de l'appel préalable à ***InitNetwork***, après exécution complète du réseau.

Dans le corps du programme, entre les appels aux fonctions ***InitNetwork*** et ***ExecuteNetwork***, il est nécessaire de spécifier les neurones utilisés au départ de l'exécution. Pour ce faire, la fonction principale offerte par la bibliothèque est la fonction **MakeNeuron**.

MakeNeuron Cette fonction permet au programmeur de définir un neurone statique (voir section 7.2.4). Elle prend en argument les fonctions caractéristiques du neurone créé et l'immatriculation de ce neurone. Elle ne peut être utilisée que dans le corps du programme, avant la demande d'exécution du réseau par **ExecuteNetwork**.

L'exemple du jeu de la vie

L'exemple 6 présente le corps du programme du jeu de la vie permettant d'initialiser une grille de 64 cellules avec les immatriculations présentées dans la figure 7.8.

Le réseau sera totalement exécuté entre les appels aux fonctions ***MakeNeuron*** et ***FreeNetwork***.

7.3.2 Fonctions de déclaration des variables locales

Avec notre simulateur, les neurones disposent d'un pointeur sur leurs variables locales. Il est nécessaire, après allocation de ce pointeur, de le spécifier à la bibliothèque afin de le récupérer à chaque cycle de vie de celui-ci. Pour ce faire la bibliothèque propose deux macros : ***InitLocVar*** et ***DecLocVar***. La première spécifie à la bibliothèque le pointeur sur la variable locale du neurone tandis que la seconde permet au neurone de récupérer ce pointeur et de retrouver ses variables locales.

InitLocVar est généralement appelée au cours de la fonction d'initialisation d'un neurone et ***DecLocVar*** au début des fonctions d'itération et de terminaison.

L'exemple du jeu de la vie

L'exemple 7 présente la spécification des variables locales d'une cellule du jeu de la vie dans sa fonction d'initialisation. L'exemple 8 montre la récupération de ces variables pour l'exécution au cours de la fonction de terminaison d'une cellule.

7.3.3 Fonctions de création et de destruction des neurones

Notre simulateur fournit deux fonctions pour créer des neurones.

```
void main()
{
 int i, j;

 /* Initialisation du nombre de processeurs et du réseau */
 /* 4 processeurs et une grille de cellule (8 x 8) = 64  */
 InitNetwork(64, 4);


 /* Création des 64 cellules                              */
 for(i = 0; i < 8; i ++) {
  for(j = 0; j < 8; j ++) {
     MakeNeuron(InitCellule, IterCellule, TermCellule, i*10+j);
   }
  }

 printf("Exécution du réseau          \n");

 /* Lancement de l'exécution du réseau                    */
 ExecuteNetwork();

 printf("C'est fini...          \n");


 /* Petit ménage                                          */
  FreeNetwork();
}
```

Exemple 6 : *Programme d'initialisation et de lancement de l'exécution du jeu de la vie.*

```
void InitCellule()
{
 typevariable *VarLocale;

 VarLocale = (typevariable *) malloc(sizeof(typevariable));

 VarLocale->liens = (canal_in *) malloc(8*sizeof(canal_in));

 /* Déclaration de (*VarLocale) comme variable de la cellule */
 InitLocVar(VarLocale);

 ...
}
```

Exemple 7 : *Spécification des variables locales d'une cellule du jeu de la vie*

```
void TermCellule()
{
 typevariable *VarLocale;

 /* Récupérer la variable locale de la cellule              */
 DecLocVar(VarLocale);

 free(VarLocale->liens);
 free(VarLocale);
}
```

Exemple 8 : *Récupération des variables locales d'une cellule du jeu de la vie*

MakeNeuron Cette fonction déclare un neurone statique, avant le début de l'exécution du réseau.

MakeNewNeuron Cette fonction déclare un neurone dynamique, c'est-à-dire qu'elle crée un neurone au cours de l'exécution du réseau. Elle est appelée par des neurones du réseau. Le nouveau neurone sera créé lors du cycle suivant cet appel.

Il existe différentes fonctions permettant la destruction d'un ou plusieurs neurones. Lors du cycle suivant l'appel à l'une des fonctions suivantes, le réseau exécute la fonction de terminaison des neurones tués, le neurone disparaît effectivement du réseau lors du cycle suivant.

kill Cette fonction détruit le neurone dont l'immatriculation est donnée en paramètre. Elle peut être appelée par tout neurone vivant du réseau.
kill_me Cette fonction détruit le neurone qui l'appelle.
kill_all Cette fonction détruit tous les neurones vivants du réseau.

Après la mort d'un neurone, les connexions sur celui-ci sont détruites, elles pointent sur ***NULL***.

7.3.4 La gestion des communications

Il existe plusieurs fonctions et macros de notre bibliothèque ayant trait aux communications dans les réseaux connexionnistes. Ces fonctions servent à manipuler les sorties, les connexions et les valeurs reçues par l'intermédiaire des canaux de communication.

Pour notre simulateur, un canal de communication est une variable de type ***canal_in***.
Pour spécifier un réseau il faut préciser, au sein des neurones, quelle est la variable de sortie, quels sont les canaux de communication et comment sont utilisées les entrées.

Les fonctions de gestion de la sortie d'un neurone

DeclareOutput Cette fonction permet au neurone appelant de déclarer l'adresse de sa variable de sortie et la fonction de copie qui lui est associée (voir exemple 9). La sortie est déclarée par

son adresse et sa fonction de copie par un pointeur de fonction. Cette fonction est généralement utilisée dans la fonction d'initialisation du neurone.

```
/* Spécification de la sortie d'une cellule                          */
/* Déclaration de la variable de sortie et de sa fonction de copie */

DeclareOutput(&(VarLocale->activite), CopieSortieCellule);
```

Exemple 9 : *Spécification de la sortie d'une cellule du jeu de la vie.*

OutputField C'est une macro permettant à un neurone un accès direct à sa sortie. Elle sert au neurone à faciliter la manipulation des champs de sa variable de sortie, si celle-ci est de type structuré.

Les fonctions de gestion des connexions

ConnectAt Permet au neurone appelant de créer un canal de communication avec un autre neurone. Elle prend en paramètre l'immatriculation du neurone auquel le neurone appelant souhaite se connecter et renvoie un canal de connexion, c'est-à-dire une variable de type ***canal_in*** (voir exemple 10). Ce canal permettra ensuite au neurone un accès en lecture à la variable de sortie du neurone connecté. Cette fonction peut être utilisée à tout moment de la vie du neurone.

```
/* Allocation du tableau de liens de la cellule          */
VarLocale->liens = (canal_in *) malloc(8*sizeof(canal_in));

/* connexion aux huit voisins de la cellule              */
/* me() renvoie l'immatriculation de la cellule          */
i = 0;
VarLocale->liens[i] = ConnectAt(me() - 1);
i ++;
VarLocale->liens[i] = ConnectAt(me() + 1);
i ++;
....
```

Exemple 10 : *Création des connexions d'une cellule du jeu de la vie.*

Deconnect Cette fonction permet de supprimer un canal de communication. Elle reçoit en paramètre le canal de communication à supprimer. Cette fonction permet l'implantation d'algorithmes d'élagage.

La gestion des variables de canaux de communication est laissée à l'utilisateur. Nous n'avons pas souhaité imposer ou masquer les variables de liens ou les tableaux de liens pour permettre plus de souplesse à l'implantation. Cela permet, par exemple, de différencier les liens feed-back des liens de rétro-propagation pour une implantation de perceptron multi-couches. Cela permet aussi de créer des structures associant chaque lien d'un neurone au poids qui lui est associé.

Les fonctions sur les entrées du neurone

InputCanal et ***InputField*** Ces macros permettent de lire à travers un canal de communication. Elles sont des simplifications d'utilisation de la fonction ***Input*** (voir annexe). ***InputCanal*** reçoit en paramètre le type de l'entrée qu'il souhaite lire et le canal par lequel il reçoit cette entrée. Elle renvoie un pointeur sur l'entrée (voir exemple 11). De la même façon ***InputField*** permet un accès direct à l'un des champs de l'entrée si celle-ci est structurée.

```
/* lecture des entrées de la cellule. Les entrées sont les activités */
/* donc de type 'int'                                                */

Somme = 0;
for (i = 0; i < 8; i ++)
 Somme = Somme + InputCanal(int, VarLocale->liens[i]);

if (Somme == 3)
     VarLocale->activité = 1;
else VarLocale->activité = 0;
```

Exemple 11 : *Lecture des entrées au sein d'une cellule du jeu de la vie.*

ReturnOrigin Cette fonction permet de connaître l'immatriculation du neurone lu par un canal de communication.

C'est l'appel aux fonctions ***ConnectAt*** et ***DeclareOutput*** par les neurones du réseau qui crée la topologie de celui-ci.

L'utilisation des canaux de communication se fait sans référence au neurone connecté, les entrées reçues par ces canaux sont utilisées comme des variables du neurone.

7.3.5 D'autres outils courants

En plus des fonctions que nous avons parcourues précédemment, deux fonctions et une variable sont souvent utilisées dans les simulations de réseaux de neurones : ***me***, ***IsAlive*** et ***MaxNeuron***.

me est une fonction qui renvoie l'immatriculation du neurone appelant.
IsAlive est une fonction qui permet de connaître l'état d'un neurone, elle renvoie 1 si le neurone est vivant et 0 s'il est mort.
MaxNeuron est une variable globale contenant le numéro de la plus grande immatriculation de neurone déclarée au moment de l'appel. Cette variable est utile pour immatriculer les neurones créés dynamiquement dans le réseau.

7.3.6 Architecture générale du code d'un réseau

Le programme d'une implantation d'un réseau de neurones à l'aide de notre simulateur se fait donc en deux phases. La première contient les fonctions caractéristiques des neurones, la seconde le programme principal de la simulation.

Description des neurones Chaque type de neurone est défini et décrit par ses trois fonctions caractéristiques : fonctions d'initialisation décrivant son premier cycle de vie, fonction de terminaison décrivant son dernier cycle de vie et fonction d'itération décrivant les autres cycles de vie du neurone. Une fonction de copie de sortie est généralement ajoutée à ces fonctions.

Le programme principal Il initialise la bibliothèque sur les caractéristiques du réseau. C'est ici que sont déclarés à la bibliothèque les neurones présents lors du premier cycle de vie du réseau. Ils sont décrits par leurs fonctions caractéristiques.

La commande ***ExecuteNetwork()*** lance l'exécution du réseau jusqu'à la fin d'activité de celui-ci : la mort de tous les neurones qui le composent.

L'exemple du jeu de la vie

Le programme de l'implantation du jeu de la vie s'écrit donc en deux phases : L'écriture des fonctions caractéristiques des cellules (exemples 13, 14 et 15) et le programme principal (exemple 12).

7.4 Les fonctions de gestion des couches

En plus des fonctions présentées en section 7.5, notre simulateur propose des outils permettant d'implanter des réseaux à couches. Ces réseaux sont en effet très présents dans la communauté et certaines de leurs spécificités peuvent être laborieuses à implanter avec les seuls outils précédents.

Il peut donc être pratique d'exploiter les propriétés des réseaux à couches tout en gardant notre postulat : définir un réseau connexionniste en définissant les neurones de ce réseau, neurones qui, en créant les connexions leur servant d'entrées, créent la topologie du réseau.

Nous proposons donc une liste de fonctions permettant de définir et de connecter des couches de neurones. Nous définissons une couche comme une somme de neurones identiques, i.e qui possèdent les mêmes fonctions caractéristiques.
Les couches du réseau sont numérotées. Ainsi un neurone déclaré comme appartenant à une couche dispose de deux identifiants distincts, son immatriculation et ses *coordonnées de couche* composées du couple numéro de sa couche, position du neurone dans la couche. Ce nouvel identifiant peut être utilisé pour demander la connexion à ce neurone, demander sa mort, etc.
Tout neurone peut demander une connexion totale à une couche du réseau. Ce neurone obtient ainsi une connexion sur chacun des neurones de la couche.
Les différentes fonctionnalités offertes par notre bibliothèque pour l'utilisation des couches sont rapidement explicitées dans le tableau de la figure 7.14

Les coordonnées d'un neurone peuvent donc être retrouvées à l'aide des fonctions ***PositionInLayer*** et ***WhichLayer*** pour un neurone souhaitant ses propres coordonnées, ***WhichLayerNeur*** et ***PosInLayerNeur*** pour les coordonnées d'un autre neurone.

7.5 Différents types de fonctions

Les fonctions proposées par notre bibliothèque peuvent être divisées en trois catégories, en fonction de leur portée temporelle. En effet nos différentes fonctions n'ont pas toutes la même portée au cours de d'une implantation, elles peuvent concerner l'implantation du réseau ou

```
typedef struct {
  canal_in    *liens;       /* Tableau de canaux de communication */
  int        activite;      /*       Activité de la cellule       */
  int      iterations;      /*       Compteur d'itérations        */
} typevariable;


void InitCellule();

void IterCellule();

void TermCellule();


void main()
{
 int i, j;

 /* Initialisation du nombre de processeurs et du réseau */
 /* 4 processeurs et une grille de cellule (8 x 8) = 64  */
 InitNetwork(64, 4);


 /* Création des 64 cellules                              */
 for(i = 0; i < 8; i ++) {
  for(j = 0; j < 8; j ++) {
     MakeNeuron(InitCellule, IterCellule, TermCellule, i*10+j);
   }
  }

 printf("Exécution du réseau          \n");

 /* Lancement de l'exécution du réseau                    */
 ExecuteNetwork();

 printf("C'est fini...          \n");


 /* Petit ménage                                          */
  FreeNetwork();
}
```

Exemple 12 : *Programme principal d'une implantation du jeu de la vie.*

```
void InitCellule()
{
 int i, j;
 typevariable *VarLocale;


 VarLocale = (typevariable *) malloc(sizeof(typevariable));

 VarLocale->liens = (canal_in *) malloc(8*sizeof(canal_in));


 /* Mise à zéro du compteur d'itérations                    */
 VarLocale->iterations = 0;

 /* Déclarer (*VarLocale) comme variable de la cellule      */
 InitLocVar(VarLocale);

 /* Déclarer &(VarLocale->activite) comme sortie du neurone */
 DeclareOutput(&(VarLocale->activite), CopieSortieCellule);

 /* Connexion aux huit cellules voisine                     */
 i = 0;
 VarLocale->liens[i] = ConnectAt(me() - 1);      i ++;
 VarLocale->liens[i] = ConnectAt(me() + 1);      i ++;

 for(j = 0; j < 3; j++) {
 VarLocale->liens[i] = ConnectAt(me() -9 -j);    i ++;
 VarLocale->liens[i] = ConnectAt(me() +9 +j);    i ++;
 }

 /* Détermination de la première activité de la cellule     */
 if (((me()%3)) == 0)
   VarLocale->activite = 1;
 else
   VarLocale->activite = 0;

}
```

Exemple 13 : *Fonction d'initialisation d'une cellule du jeu de la vie.*

```
void IterCellule()
{
 int Somme;
 typevariable *VarLocale;

 /* Récupérer la variable locale de la cellule             */
 DecLocVar(VarLocale);

 /* Effectuer la somme des variables reçues par les entrées */
 Somme = 0;
 for (i = 0; i < 8; i ++)
  Somme = Somme + InputCanal(int, VarLocale->liens[i]);

 /* Evaluer l'activité de la cellule                       */
 if (Somme == 3)
    VarLocale->activité = 1;
 else
    VarLocale->activité = 0;

 /* Condition de terminaison                               */
 /* La cellule demande à mourir au terme de 800 itérations */
 if((VarLocale->iterations) == 800)
   kill_me();
 else
   VarLocale->iterations ++;

}
```

Exemple 14 : *Fonction d'itération d'une cellule du jeu de la vie*

```
void TermCellule()
{
 typevariable *VarLocale;

 /* Récupérer la variable locale de la cellule             */
 DecLocVar(VarLocale);

 free(VarLocale->liens);
 free(VarLocale);
}
```

Exemple 15 : *Fonction de terminaison d'une cellule du jeu de la vie*

InitLayers	Fonction permettant au programmeur de définir le nombre de couches de son réseau.
DecLayer	Fonction utilisée pour déclarer une couche.
ConnectNeuronInLayer	Fonction permettant au neurone appelant de se connecter à un neurone en précisant sa couche et la place qu'il occupe sur celle-ci.
ConnectAtLayer	Fonction permettant à un neurone de se connecter à tous les neurones de la couche spécifiée.
kill_NeuroneInLayer	Fonction demandant la mort d'un neurone en utilisant ses coordonnées de couche.
WhichLayer	Fonction retournant le numéro de la couche à laquelle appartient le neurone appelant.
PositionInLayer	Fonction retournant la place du neurone appelant sur sa couche.
WhichLayerNeur	Fonction renvoyant, à partir de l'immatriculation d'un neurone, le numéro de la couche le contenant.
PosInLayerNeur	Fonction renvoyant, à partir de l'immatriculation d'un neurone, la position de ce neurone sur sa couche.
SizeOfLayer	Fonction retournant la taille, le nombre de neurones vivants, d'une couche donnée.
NbLayers	Variable globale du réseau contenant le nombre de couches déclarées dans le réseau.

FIG. 7.14 – *Tableau des fonctions de gestion des couches de neurones.*

seulement celle d'un neurone. De plus, la division par cycle des exécutions distingue alors les fonctions selon qu'elles s'appliquent immédiatement lors de leur appel ou avec un décalage.

7.5.1 Les fonctions hors exécution du réseau

Ces fonctions sont utilisées dans la partie principale du programme. Elles servent à initialiser et désallouer les différentes variables de notre bibliothèque, à définir les neurones présents dans le réseau de départ de l'exécution.
Ces fonctions sont les suivantes :

InitNetwork()	***MakeNeuron***()	***ExecuteNetwork***()
InitLayers()	***DecLayer***()	***FreeNetwork***()

7.5.2 Les fonctions à effet immédiat

Ces fonctions sont exécutées au cours du cycle de leur appel et renvoient une valeur. A l'exception de ***Deconnect*** ces fonctions n'agissent pas sur le réseau. Soit elles renvoient une variable du réseau, soit elles spécifient une variable au réseau.
Elles sont les suivantes :

Deconnect()	***Input***()	***SizeOfLayer***()
InitLocVar()	***InputCanal***()	***WhichLayer***()
DecLocVar()	***InputField***()	***PositionInLayer***()
DeclareOutput()	***OutputField***()	***WhichLayerNeur***()
me()	***ReturnOrigin***()	***PosInLayerNeur***()
IsAlive()		

7.5.3 Les fonctions décalées d'un cycle

Ce sont les fonctions appelées au cours d'un cycle mais qui n'ont d'effet sur le réseau que lors du cycle suivant l'appel. Ce sont des fonctions qui agissent sur la topologie du réseau : elles ajoutent des connexions ou des neurones ou tuent des neurones.
Ces fonctions sont les suivantes :

ConnectAt()	***kill_all***()	***kill_NeuroneInLayer***()
ConnectAtLayer()	***kill***()	***MakeNewNeuron***()
ConnectNeuronInLayer()	***kill_me***()	

7.6 Options de compilation

Pour utiliser au mieux notre bibliothèque, nous proposons diverses options de compilation. L'utilisation du parallélisme hardware signifie, le plus souvent, une recherche de performances. Dans le but de respecter ce critère, nous avons allégé le code de base d'un programme implanté à l'aide de notre bibliothèque. Il devient donc utile de préciser certains besoins spécifiques lors de la compilation.
Nous proposons par exemple les options présentées dans le tableau de la figure 7.15

MONO	Option spécifiant la compilation du programme sur machine séquentielle classique. La bibliothèque définira alors un code en ANSI-C. Les spécificités inhérentes à l'utilisation d'une machine parallèle MIMD sont supprimées et le programme généré pourra s'exécuter sur plate-forme séquentielle.
BNDEBUG	Option spécifiant la compilation du programme en mode *debug*. Avec cette option, l'exécution du programme sera complétée de divers tests entraînant les messages d'erreurs.

FIG. 7.15 – *Tableau des options de compilation proposées par le simulateur.*

Utiliser une option comme **BNDEBUG** ralentit les applications car elle ajoute de nombreux tests lors de l'exécution du réseau. En effet, en exécution avec l'option **BNDEBUG**, la bibliothèque vérifie les allocations mémoire et différents points spécifiques au connexionnisme. Elle vérifie, par exemple, que les canaux utilisés sont valides, que les neurones connectés sont vivants ou qu'un neurone dont la sortie est lue a effectivement déclaré une sortie. En cas de problème, la bibliothèque envoie un message d'erreur qui se voudrait explicite. Hors utilisation de cette option, les erreurs ne sont pas détectées lors de l'exécution et peuvent déclencher des erreurs fatales ou des incohérences lors de l'exécution.

L'option **BNDEBUG** est dédiée au *debuggage* des implantations et doit être supprimée pour les exécutions sous peine de grosses pertes de performances.

7.7 Un bilan sur le simulateur

Nous avons présenté notre simulateur de réseaux de neurones. Sa particularité est qu'il décrit et décompose les réseaux connexionnistes au niveau de leurs neurones.

Ce simulateur se présente sous la forme d'un faible nombre de fonctions sur le langage C, faible nombre devant rendre plus accessible l'utilisation de la bibliothèque. Les fonctions proposées ont pour unique ambition de décrire les réseaux et les neurones composant ces réseaux. Les autres aspects des implantations se font à l'aide des outils, nombreux et efficaces, offerts par le langage C et ses extensions.

La construction des réseaux basée sur les neurones ne limite pas le type des neurones pouvant être simulés et, par là-même, permet d'implanter toute sorte de réseaux connexionnistes. Si l'implantation de réseaux construits autour de neurones formels ne pose pas de problèmes (un neurone formel peut se simuler avec un type de variable locale tel que présenté dans l'exemple 16), le peu de limites dans la construction des neurones permet de simuler des modèles plus complexes ou des modèles distribués de nature quelque peu différente, comme nous l'avons montré avec le jeu de la vie. Il est en effet possible d'ajouter tout type de variable à l'exemple 16 et d'augmenter d'autant les potentialités des neurones construits.

Il est aussi possible de construire des modèles moins évidents pour notre simulateur. L'exemple 17 montre comment pallier le fait que notre bibliothèque ne permet que des connexions à sens unique, tandis que le perceptron multi-couches et son apprentissage par rétro-propagation du gradient nécessitent des connexions à double sens. Il est ainsi possible de doubler la sortie des

```
typedef struct {
  canal_in   *entrees;   /*    Connexion avec les autres neurones    */
  float       *poids ;   /*          Poids des connexions            */
  int          Sortie;   /*      Valeur d'activation du neurone      */
} typevariableNeuroneFormel;
```

Exemple 16 : *Type de variable locale d'un neurone formel*

neurones, les neurones connectés utiliseront ensuite la sortie correspondant à leurs besoins. Ils utiliseront le champ `activation` de la sortie pour la phase de détermination de l'activation, et le champ `erreur` de la sortie pour la phase de rétro-propagation.

```
typedef struct {
   float activation;     /*    Propagation de l'activité du neurone    */
   float erreur;         /*       Rétropropagation de l'erreur         */
} TypeSortie;

typedef struct {
  canal_in   *entrees;   /*    Connexion avec les autres neurones      */
  float       *poids ;   /*          Poids des connexions              */
  TypeSortie   Sortie;   /*      Valeur d'activation du neurone        */
} typevariableNeuroneFormel;
```

Exemple 17 : *Type de variable locale d'un neurone de perceptron multi-couches*

Notre simulateur permet aussi, avec ses fonctions de création et de destruction dynamique des neurones et des connexions de simuler des réseaux évolutifs.
De même des réseaux asynchrones, comme certains modèles de Hopfield, peuvent être implantés avec notre simulateur. Si la bibliothèque évalue tous les neurones du réseau à chaque cycle d'exécution de celui-ci, les neurones peuvent ne rien exécuter durant certains cycles de vie. Nous verrons avec les exemples du chapitre 9 que les fonctions d'itération des neurones sont souvent implantées sur plusieurs cycles de vie, avec parfois des cycles sans action de la part d'une partie des neurones.

Avant de présenter les exemples d'implantations et de performances obtenues avec notre simulateurs, nous présentons dans le chapitre suivant les aspects les plus importants de l'implantation, effectuée par la bibliothèque, et des exécutions sur les machines parallèles.

Chapitre 8

L'implantation de la bibliothèque sur machines parallèles

Les réseaux implantés à l'aide de notre bibliothèque peuvent être exécutés, au choix de l'utilisateur, sur ordinateur séquentiel classique comme sur ordinateur parallèle de type MIMD à mémoire partagée. La bibliothèque se charge totalement du passage sur machine parallèle. Aucune modification du programme implanté n'est nécessaire pour cette exécution. L'utilisateur doit seulement compiler son programme sur un ordinateur parallèle de type MIMD à mémoire partagée.

Pour permettre les exécutions parallèles, la bibliothèque doit gérer les spécifications permettant d'obtenir des performances, en terme d'accélérations des applications, sur ce type de machine.

Notre bibliothèque se charge donc, pour l'utilisateur, des problèmes de placement des neurones sur les processeurs, de placement des données sur les différentes mémoires, des indispensables communications entre les différents processeurs et des synchronisations entre ces processeurs.

Nous allons présenter, dans cette partie du manuscrit, la méthode utilisée par la bibliothèque pour installer et exécuter un réseau de neurones artificiels, quel qu'il soit, sur un réseau de processeurs (voir figure 8.1).

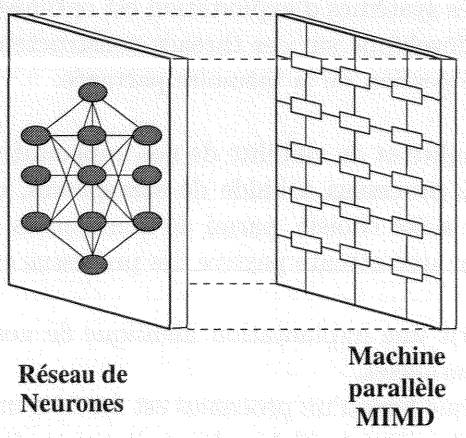


FIG. 8.1 – *Notre objectif : Exécuter les réseaux de neurones artificiels simulés à l'aide de notre bibliothèque sur un réseau de processeurs.*

8.1 Les choix d'implantation

8.1.1 Les limites de spécification

Nous avons choisi d'effectuer nos implantations parallèles sur ordinateurs parallèles MIMD à mémoire partagée. Comme nous l'avons vu précédemment (chapitre 4), la technologie la plus utilisée actuellement pour ce type de machine parallèle est la technologie à mémoire partagée distribuée cc-NUMA.

Nos implantations et expérimentations se font, pour des raisons de disponibilité géographique, sur une Origin2000 (architecture développée par Silicon Graphics Inc). Cette architecture parallèle dispose de nœuds bi-processeurs reliés entre eux par une topologie en hypercube. Chaque processeur de cette machine partage physiquement l'espace mémoire avec son voisin de nœud et dispose de deux niveaux de cache. L'utilisation de ces spécificités topologiques peut permettre d'optimiser les applications et d'obtenir ainsi de meilleures performances.

Pour des raisons de portabilité et de pérénité de notre simulateur, nous avons choisi de limiter les spécifications techniques de notre bibliothèque à la technologie DSM cc-NUMA (voir page 62). Les acheminements des données d'une mémoire de la machine à une autre et la gestion des niveaux supplémentaires de cache mémoire sont laissés aux bons soins du système d'exploitation de la machine. Nos efforts d'implantation doivent seulement permettre à ce système d'exploitation de gérer au mieux ces spécificités techniques.

Pour permettre une plus grande portabilité de la bibliothèque, les fonctions dédiées à l'architecture, création de la mémoire partagée, création des threads et outils de synchronisation, sont centralisées et peu déterminantes. Elle sont, par là-même, simples à remplacer par leurs fonctions équivalentes, pour adapter la bibliothèque à une nouvelle machine.

Notre bibliothèque est donc développée pour s'adapter à toutes les architectures matérielles de type DSM cc-NUMA.

8.1.2 Les outils de gestion de la mémoire partagée

Pour implanter notre bibliothèque en mémoire partagée, nous avons choisi d'utiliser des threads constructeurs. Ces *processus légers* sont fournis sur toutes les architectures à mémoire partagée et leur gestion par les systèmes d'exploitation est optimisée sur ces architectures. L'avènement actuel de OpenMP, développé sur ces threads constructeurs, nous garantit une certaine pérénité de cette méthode de gestion de la mémoire partagée.

Pour les raisons de portabilité et de viabilité de nos implantations, nous avons développé nos outils de synchronisation des processus à l'aide de *sémaphores*, outils fournis en standard sur le langage C. Cette méthode a été choisie, parmi de nombreuses autres [Kufrin, 1998], pour sa généricité et se caractérise par une attente passive des processus et l'existence de zones protégés au sein des barrières.

La généricité nous garantit une implantation identique de nos barrières de synchronisation sur toutes les architectures parallèles.

L'attente passive signifie que lorsqu'un processus est arrêté à une barrière de synchronisation, il s'endort jusqu'à ce que la barrière le libère. Ainsi, l'attente d'un processus ne charge pas le processeur qui le porte. Cette propriété est intéressante sur les machines parallèles actuelles car elles sont fréquemment partagées entre plusieurs utilisateurs et parfois surchargées. Nos barrières de synchronisation permettent, dans ce cas, d'éviter aux applications d'être ralenties par des processus en attente.

De plus notre implantation de barrières de synchronisation par sémaphores offre une période pendant laquelle un seul processus est actif. Cette zone protégée nous permet de mettre à jour sans risque les variables globales de notre bibliothèque.

8.2 Les contraintes liées à la mémoire partagée

8.2.1 L'équilibrage de charge

Nous avons opté pour un équilibrage de charge naïf. Les neurones sont des unités à faibles capacités calculatoires et à faible coût en temps de calcul. Le poids en temps de calcul est donc une conséquence du grand nombre de neurones des réseaux. Ces propriétés des réseaux connexionnistes nous laissent penser que la différence de coût en temps de calcul entre deux types d'unités du réseau sont négligeables par rapport au temps d'exécution d'un réseau.

Nous avons choisi de répartir équitablement les neurones sur les différents processeurs utilisés par la bibliothèque.

Pour la gestion des réseaux dynamiques, les processeurs sont dotés d'un compteur du nombre d'*unités vivantes* gérées par celui-ci. Les nouvelles unités créées par le réseau seront donc exécutées par le processeur ayant la plus petite charge.

Notre bibliothèque ne propose pas de protocoles de correction de l'équilibrage de charge. Si la charge se déséquilibre fortement, c'est-à-dire si un processeur du réseau gère beaucoup plus de neurones qu'un ou plusieurs autres processeurs, notre bibliothèque n'effectue pas de nouvelles répartitions des neurones pour équilibrer la charge.

Pour effectuer cette correction, il serait nécessaire d'effectuer de nombreux tests sur les différentes charges, puis de transférer des neurones d'un processeur à un autre. Ces transferts impliquent de nombreuses communications entre les processeurs et une mise à jour des tables de gestion des neurones de notre bibliothèque. Ces opérations sont donc extrêmement coûteuses en temps.

Le risque de déséquilibrage étant réduit à un très faible nombre de topologies connexionnistes, en fait aux algorithmes d'élagage purs (sans aucune création de neurones), nous avons choisi, dans un premier temps, de ne pas développer cette option.

8.2.2 La gestion des caches

Avec les ordinateurs à mémoire partagée cc-NUMA, tout processeur dispose d'un accès direct à la mémoire de tous les processeurs du système. Lorsqu'un processeur a besoin d'une donnée, il copie préalablement cette donnée dans son cache mémoire. Pour ce faire, le système copie la ligne de cache[30] complète qui contient la donnée demandée par le processeur.

De plus, quand un processeur modifie une donnée, le système invalide toutes les copies de la ligne de cache contenant cette donnée. Si un autre processeur utilise une variable de cette ligne, la variable modifiée ou une autre, il doit préalablement recharger la ligne de cache dans son propre cache. Ces spécificités techniques entraînent deux difficultés principales dans la recherche de performances, *les conflits de cache* et *les faux partages de cache*. Ces erreurs de cache peuvent ralentir considérablement les applications au point de rendre l'application plus lente sur plusieurs processeurs qu'en séquentiel.

30. Selon les systèmes, les copies peuvent se faire par lignes ou par pages.

Les conflits de cache

Il y a conflit de cache lorsqu'un processeur modifie fréquemment une donnée et que d'autres processeurs lisent souvent cette même donnée (voir figure 8.2). Ainsi ces processeurs doivent recharger cette donnée après chaque modification. Ce qui ralentit d'autant les exécutions.

Pour pallier cette difficulté il est donc nécessaire de limiter au mieux les données lues par des processeurs et modifiées par un ou plusieurs autres processeurs du système.

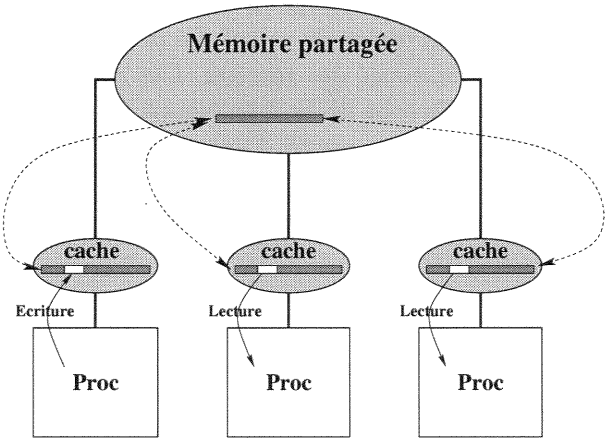


FIG. 8.2 – *Conflit de cache. Une même variable est modifiée par un processeur tandis qu'elle est lue par d'autres. Après modification de la variable, chaque lecture nécessite un rechargement de la ligne de cache.*

Le faux partage de cache

Un mauvais partage de cache survient quand plusieurs processeurs du système modifient des données différentes (voir figure 8.3), mais placées sur une même ligne de cache. Dans ce cas les processeurs sont contraints de recharger la ligne de cache après chaque modification effectuée par l'autre processeur, pour des opérations internes à chaque processeur.

Pour éviter ce problème il est nécessaire de s'assurer, si possible, de l'exclusivité d'un processeur sur les lignes de cache. Il faut donc s'assurer que les valeurs partageant une même ligne de cache sont modifiées par un même processeur.

8.2.3 La communication entre les processeurs

Comme nous l'avons vu précédemment, l'utilisation d'une machine parallèle nécessite, pour le programmeur, de gérer les communications entre les différents processeurs utilisés de la machine. Théoriquement, l'utilisation de la communication par partage de mémoire simplifie cette étape de la programmation, chaque processeur ayant accès, en lecture et en écriture, à l'espace mémoire complet de la machine.

Toutefois, tout transfert de données à travers la machine est coûteux en temps. Il est donc indispensable, pour espérer des performances, de distinguer les données utilisées exclusivement par un processeur et celles manipulées par plusieurs processeurs. Les premières doivent être regroupées sur le cache du processeur qui les utilise ou sur un segment de mémoire proche de ce

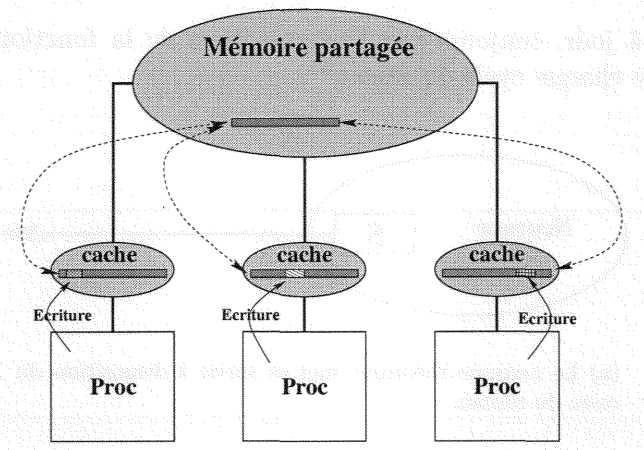


FIG. 8.3 – *Faux partage de cache. Des données différentes et chacune modifiée par un processeur différent de la machine partagent une même ligne de cache.*

processeur. Les secondes doivent être stockées dans des segments de mémoire ne contenant que des données partagées. Ainsi, les lignes de caches communiquées d'un processeur à un autre ne contiendront que des données partagées.

Pour gérer les communications entre les processeurs, la bibliothèque distingue les zones de mémoire dédiées à un processeur et les zones de mémoire effectivement partagées. Les premières contiendront les données manipulées par un processeur unique et placées, sur DSM, sur une mémoire physique proche du processeur qui les manipule. Les secondes contiennent les données manipulées par différents processeurs au cours d'une exécution. De plus, pour éviter les erreurs de cache, nous nous assurons que deux variables manipulées par deux processeurs différents seront placées sur des lignes de cache distinctes. Nous éliminons aussi toutes les périodes pendant lesquelles un processeur modifierait une donnée lue par d'autres processeurs.

8.3 La gestion des connexions entre neurones

La création des connexions, avec notre simulateur, se fait en deux phases distinctes :

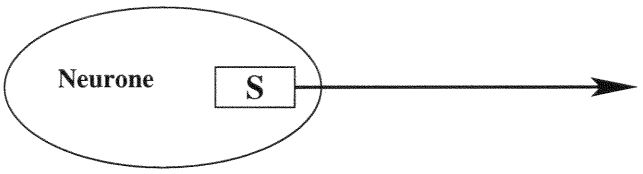
1. Les neurones créent leurs sorties.
2. Les neurones se connectent aux sorties de leurs condisciples.

Ces deux phases sont traitées de la même manière en exécution séquentielle et parallèle. La différence entre les deux exécutions provient de la gestion en mémoire des différentes variables concernées. Si l'exécution séquentielle ne pose aucun problème de mémoire, la gestion de cette mémoire est l'un des points principaux pour l'obtention de performances sur machines parallèles.

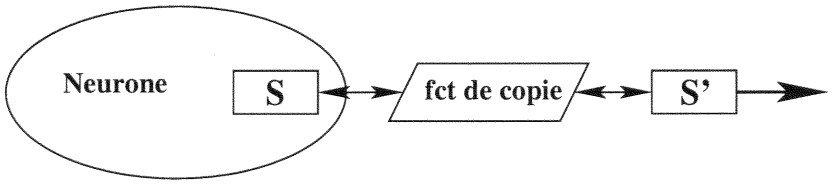
8.3.1 La création des sorties

A la création de la sortie, la bibliothèque utilise la *fonction de copie* qui lui est associée pour créer un clone de cette sortie. C'est ce clone qui pourra ensuite être lu par les autres neurones du réseau (voir figure 8.4). La sortie du neurone peut ainsi être manipulée comme une variable normale du neurone.

Ce clone est mis à jour, toujours par l'intermédiaire de la fonction de copie fournie par l'utilisateur, à la fin de chaque cycle du réseau[31].



(a) Le neurone théorique met sa sortie à disposition du reste du réseau.



(b) La simulation crée un clone de la sortie pour les communications.

FIG. 8.4 – *Pour simuler l'accès à la sortie d'un neurone, figure 8.4(a), la bibliothèque utilise la fonction de copie de sortie du neurone pour créer un clone de cette sortie, figure 8.4(b).*

8.3.2 Les demandes de connexions

Lorsqu'un neurone demande la connexion à un autre neurone, la bibliothèque crée un canal de communication entre ces deux neurones. En fait, la bibliothèque crée un couple `(n°canal, adresse)`, associant le canal nouvellement construit à l'adresse **du clone** de la sortie du neurone demandé.

Chaque neurone possède donc un tableau de couples `(n°canal, adresse)`. Ce tableau permet à la bibliothèque de transmettre la sortie *valide* au neurone quand il demande à lire à travers son canal de connexion.

8.3.3 L'implantation parallèle des communications

Les sorties des neurones sont les seules variables transmises d'un neurone à l'autre. En conséquence, l'ensemble des variables de sortie des neurones gérés par un processeur est la réunion des variables partagées d'un neurone.

Les variables de sortie accessibles étant (voir section 8.3.1) les clones de ces variables, c'est donc l'ensemble des variables *clones des sorties* qui représente les variables partagées d'un processeur.

Les sorties, variables déclarées par les neurones, sont donc localisées dans les espaces mémoires réservés aux processeurs (voir section 8.2.3). Les clones, pour leur part, sont créés dans

31. Pour limiter la durée de mise à jour du clone, l'exécution de la fonction de copie se fait sans les appels aux allocations incluses dans la fonction, allocations chères en temps de réponse.

la partie effectivement partagée de la mémoire.

La gestion des clones se fait, au cours d'un même cycle d'exécution d'un réseau connexionniste, en deux phases : la phase de calcul d'un cycle du réseau et la phase de mise à jour des clones.

Pendant la phase de calcul du réseau, les neurones utilisent les clones *en lecture* (ce sont leurs entrées) pour calculer leur nouvelle sortie (voir figure 8.5). Les processeurs manipulent donc l'espace de mémoire partagé uniquement en lecture.

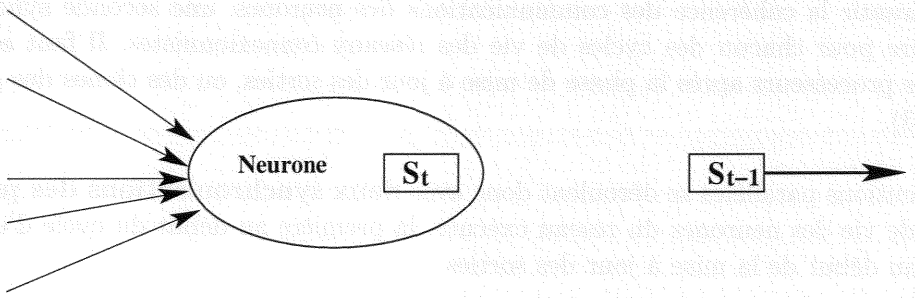


FIG. 8.5 – *En phase de calcul du réseau, chaque neurone calcule sa sortie. Le clone est disponible en lecture et possède la valeur de la sortie au cycle précédent.*

En phase de mise à jour des clones, les processeurs copient les sorties des neurones sur leurs clones respectifs. Lors de cette phase, chaque processeur manipule *en écriture* les clones appartenant à ses propres neurones (voir figure 8.6). Pour éviter les défauts de cache, chaque processeur dispose donc d'un espace propre dans la partie de la mémoire effectivement partagée. Ainsi, lors de la phase de mise à jour des clones[32], chaque processeur modifie son propre espace de mémoire partagée.

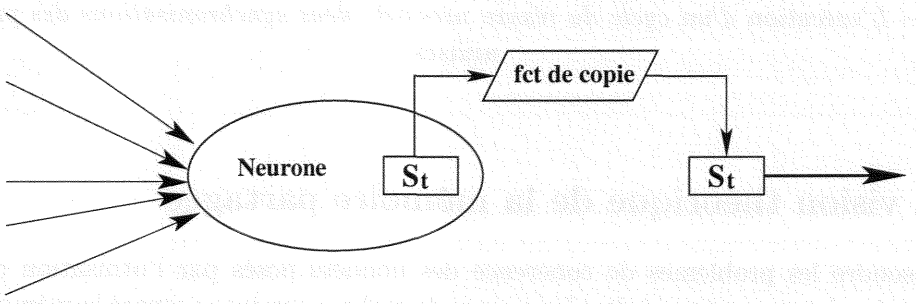


FIG. 8.6 – *En phase de mise à jour des clones, le processeur utilise la fonction de sortie de chaque neurone pour recopier la nouvelle valeur du clone de la sortie.*

Pour des raisons de cohérence des calculs, ces deux phases sont distinctes et séparées par une barrière de synchronisation.

8.4 La synchronisation entre les processeurs

L'utilisation d'une machine parallèle nécessite la gestion des synchronisations entre les processeurs utilisés pour les exécutions. Les synchronisations gérées par notre bibliothèque, implantées

32. Cette phase est aussi appelée *phase de mise à jour des sorties.*

à l'aide de sémaphores (voir section 8.1.2), doivent garantir la bonne synchronisation des réseaux de neurones implantés et la cohérence des variables partagées par les différents processeurs.

Pour obtenir une synchronisation par cycles des neurones simulés, les processeurs utilisés lors de l'implantation parallèle sont eux-mêmes synchronisés par cycles. C'est-à-dire que les différents processeurs exécutent un cycle de vie de chacun des neurones qu'ils gèrent avant de tous se synchroniser pour commencer le cycle suivant.

Pour garantir la cohérence des communications des neurones, une seconde synchronisation est nécessaire pour chacun des cycles de vie des réseaux connexionnistes. Il faut en effet synchroniser les processeurs après la phase de mise à jour des sorties, ou des clones des sorties (voir section 8.3.3).

Nos exécutions parallèles se déroulent donc avec **deux synchronisations des processeurs par cycle** de vie des neurones du réseau exécuté, la première au début du cycle d'exécution et la seconde au début de la mise à jour des sorties.

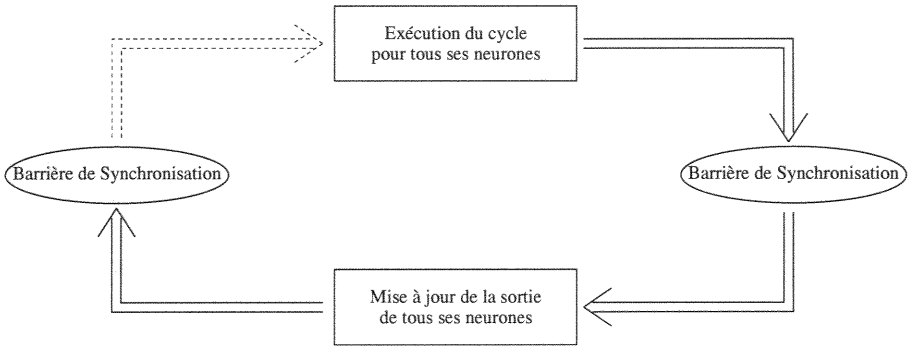


FIG. 8.7 – *L'exécution d'un cycle du réseau nécessite deux synchronisations des processeurs utilisés.*

8.5 La vision théorique de la mémoire partagée

Pour résoudre les problèmes de cohérence des données posés par l'utilisation de machines parallèles à mémoire partagée et éviter les erreurs de cache, nous avons séparé la mémoire partagée en plusieurs parties distinctes (voir figure 8.8). Ces découpages ne sont que le reflet de notre gestion de la mémoire, technologiquement tout l'espace mémoire est partagé entre les processeurs.

Les zones de mémoire privées des processeurs Il en existe une par processeur utilisé pour l'exécution. Elles contiennent les variables des processeurs qui ne sont pas partagées avec d'autres processeurs de la machines (voir section 8.2.3). Pour un processeur donné, cette zone contient toutes les données allouées par les neurones qu'il exécute.

Les zones de mémoire partagées des processeurs Il en existe une par processeur utilisé. Ce sont des zones effectivement partagées par l'exécution. Les données placées dans ces zones sont lues par tous les processeurs de la machine. Mais les données placées dans ces zones ne sont modifiées que par le processeur propriétaire de la zone. Ces zones contiennent les clones des

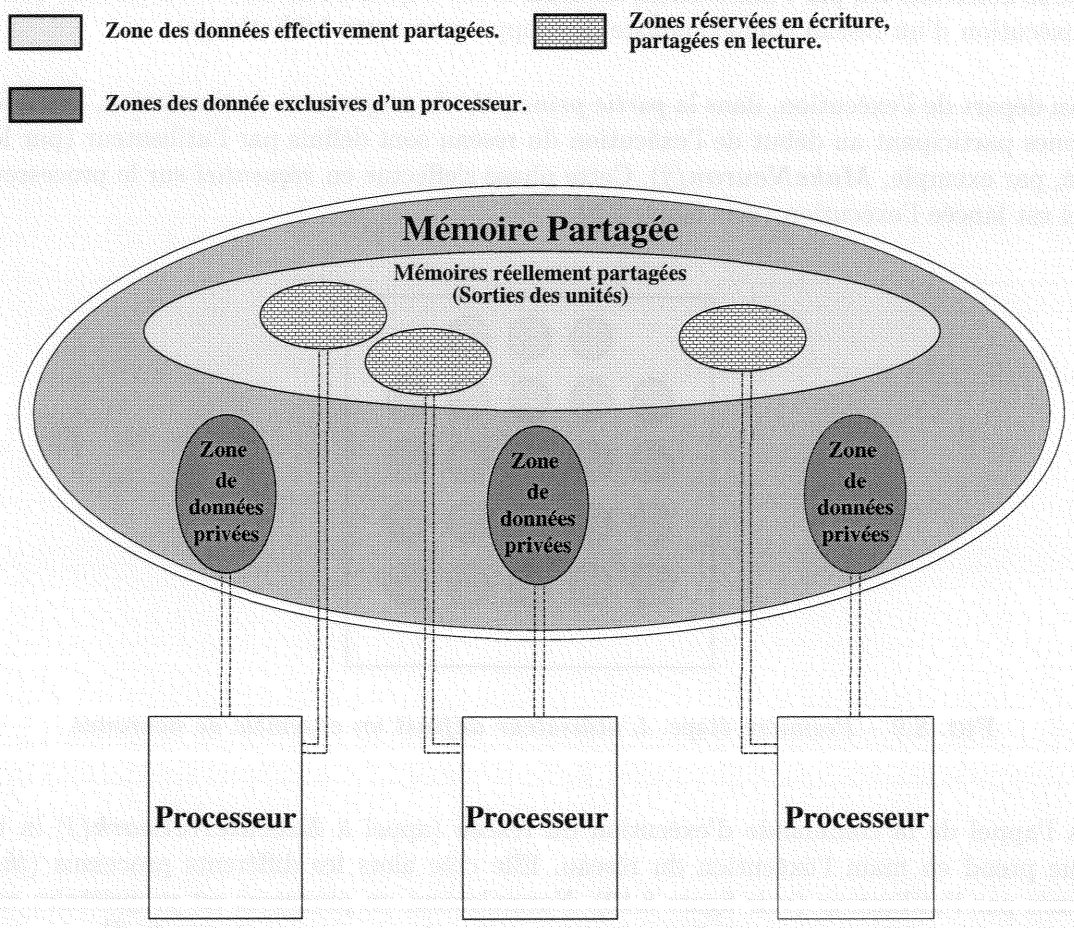


FIG. 8.8 – *Notre gestion de la mémoire. La mémoire partagée est scindée en deux parties. Une première contenant les variables non partagées par les différents processeurs. Une seconde contenant les variables partagées, accessibles en lecture à tous les processeurs. Cette dernière est séparée en zones de données modifiables par un unique processeur.*

sorties des neurones placés sur le processeur. Ces clones sont les sorties effectivement lues par les neurones.

8.6 Les différentes étapes de l'exécution

8.6.1 L'exécution sur la machine parallèle

Nous décrivons ici, par l'illustration, les différentes étapes, au niveau de la machine parallèle, de l'exécution d'un réseau connexionniste développé avec notre bibliothèque.

Au départ de l'exécution, dans la *partie principale du programme* de simulation, les différents neurones participant au début de l'exécution du réseau sont définis par l'utilisateur (par les appels à, par exemple, ***MakeNeuron()***). Cette phase s'effectue en *séquentiel* sur le processeur sur lequel est lancée l'exécution (voir figure 8.9).

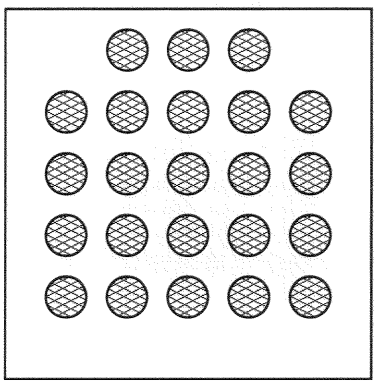


FIG. 8.9 – *Première étape. L'utilisateur* ***définit*** *un ensemble de neurones.*

A l'appel de la commande d'exécution du réseau (appel à ***ExecuteNetwork()***) la bibliothèque prend en main l'exécution du réseau. Elle crée alors les différents processus (*threads*) réclamés par l'utilisateur (voir figure 8.10). Normalement un processus est propriétaire exclusif d'un processeur et l'exécution s'effectue sur le nombre de processeurs demandés[33].

Les neurones demandés sont répartis sur les différents processeurs (voir section 8.2.1 et figure 8.11). Si les neurones ont été déclarés à l'aide des fonctions de couche, la bibliothèque répartit les différents neurones d'une couche sur les différents processeurs. Les réseaux à couches étant généralement évalués par couche, cette manipulation évite que tous les neurones d'une même couche soient exécutés par un même processeur. Dans ce cas ce processeur serait le seul actif au cours des cycles d'évaluation de la couche qu'il gère.

La bibliothèque exécute enfin le réseau. Chaque processeur exécute les fonctions caractéristiques de leurs neurones. Les créations de sorties et de connexions des différents neurones (appels à ***DeclareOutput()*** et ***ConnectAt()***) créent la topologie du réseau de neurone simulé (voir figure 8.12). Durant l'exécution, des neurones et des connexions peuvent être créés et d'autres

33. Il est ainsi possible d'utiliser le terme processeur pour parler des processus.

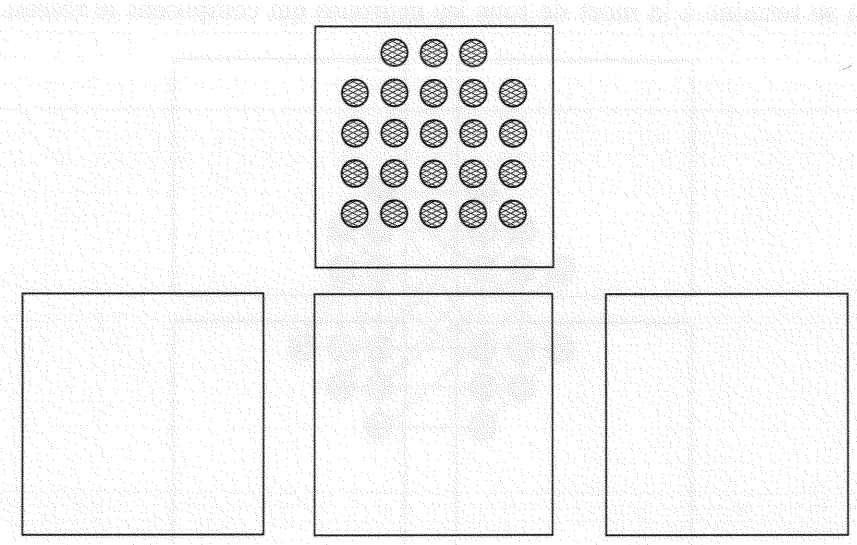


FIG. 8.10 – *Seconde étape. La bibliothèque crée les processus permettant l'exécution parallèle.*

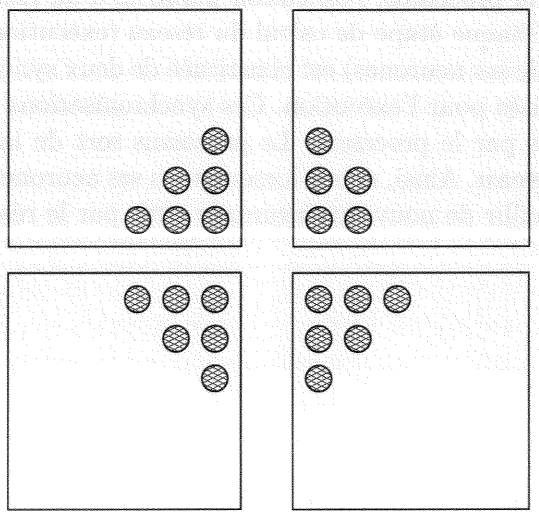


FIG. 8.11 – *Troisièmes étape. Les neurones demandés sont répartis équitablement sur les différents processus.*

détruits.

L'exécution se termine à la mort de tous les neurones qui composent le réseau.

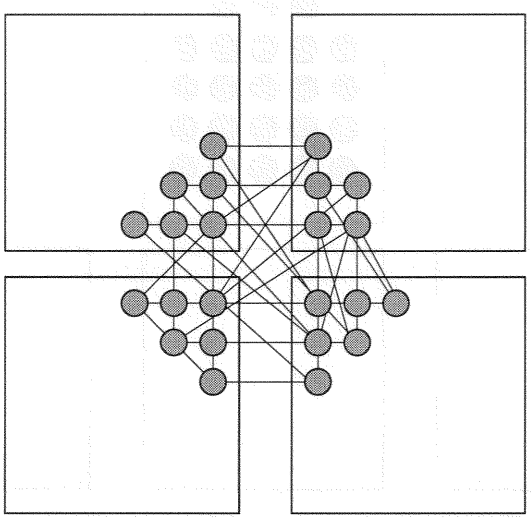


FIG. 8.12 – *Quatrième étape. La bibliothèque lance effectivement l'exécution du réseau de neurones artificiels. Les neurones sont alors* ***créés*** *et* ***exécutés*** *sur leur processeur hôte. Durant l'exécution par cycle des neurones du réseau, la topologie du réseau de neurones est construite.*

8.6.2 L'exécution des processus

Au niveau des différents processus, l'exécution parallèle d'un réseau se déroule comme présenté dans la figure 8.13. Chaque étape de calcul du réseau (exécution d'un cycle d'une fonction caractéristique de chacun de ses neurones) est constituée de deux synchronisations avec les autres processus/processeurs utilisés pour l'exécution. Ces synchronisations protègent la phase de mise à jour des neurones portés par le processus. Le processus sort de la boucle lors de la mort de **tous les neurones du réseau**. Ainsi, même lorsque tous ses neurones sont morts, un processeur peut potentiellement accueillir de nouveaux neurones créés par le réseau.

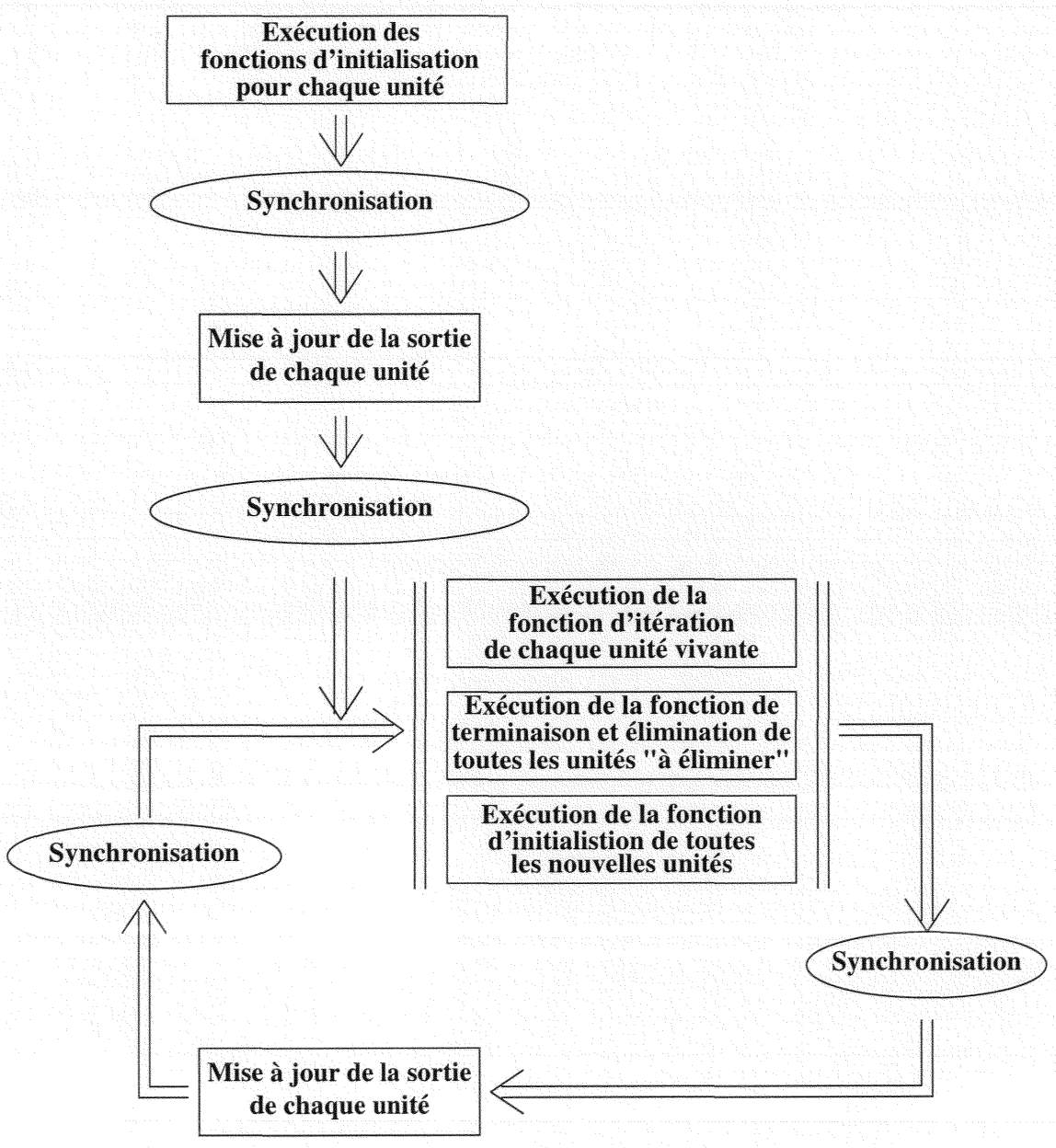


FIG. 8.13 – *Les différentes étapes de l'exécution d'un réseau, au niveau d'un processus.*

Chapitre 9

Exemples d'implantations et performances

Pour illustrer les potentialités de la bibliothèque, nous présentons dans ce chapitre trois exemples d'implantation de réseaux connexionnistes à l'aide de notre bibliothèque de simulation et les performances obtenues sur machine parallèle avec ces implantations. Nous présentons une implantation d'une carte auto-organisatrice de Kohonen ([Kohonen, 1989] et page 33), d'un réseau *growing neural gas* ([Fritzke, 1995] et page 36) et d'un réseau contextuel utilisant des OWE ([Pican, 1995], page 30), réseau à couches proche du perceptron multi-couches intégrant une dimension contextuelle dans le calcul des poids.

Ces architectures vont nous permettre d'étudier l'implantation et les performances parallèles d'un réseau classique avec la carte de Kohonen, d'une architecture évolutive avec le modèle *growing neural gas* puis des réseaux à couches et leur élagage avec l'architecture OWE.

Les performances présentées ont été obtenues sur une *Origin2000* de SGI, machine parallèle de type MIMD à mémoire partagée distribuée. La machine sur laquelle nous avons effectué nos mesures est dotée de 64 processeurs répartis sur des cartes bi-processeurs connectées en hypercube, cette machine possède 8 Go de mémoire classique et 4 Mo de mémoire cache. Les performances sont mesurées sur la machine en mode "plusieurs utilisateurs", mode fréquent d'utilisation pour ce type de machines. Les mesures sont effectuées sur le temps `utilisateur` rendu par le système (à l'aide de mesure de temps externe à l'application, mesures fournies par le système d'exploitation).

Nous détaillerons précisément les phases d'implantation de la carte de Kohonen et, la méthodologie étant acquise, nous passerons plus rapidement sur les deux modèles suivants.

9.1 L'implantation d'une carte auto-organisatrice de Kohonen

Comme nous l'avons précédemment vu, une carte auto-organisatrice se présente, comme montré dans la figure 9.1, sous la forme d'une topologie précise, grille de une à trois dimensions le plus souvent, permettant de représenter, dans un espace de faible dimension, des entrées d'un espace de plus grande dimension.

9.1.1 L'algorithme classique

L'algorithme séquentiel de l'apprentissage d'une carte auto-organisatrice se déroule comme décrit dans l'algorithme 9.1. Pour chacun des exemples présentés au réseau, l'apprentissage se

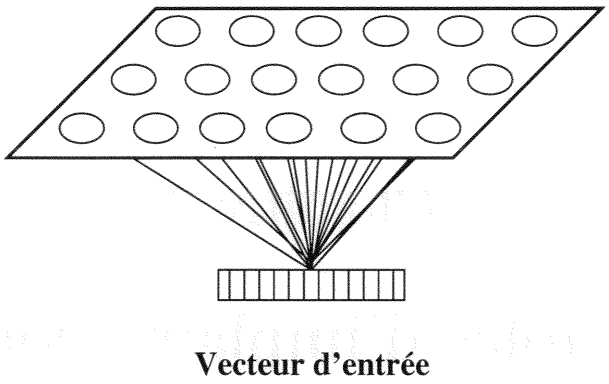


FIG. 9.1 – *Une carte auto-organisatrice de Kohonen sous forme d'une grille à deux dimensions.*

déroule en trois phases.

Algorithme 9.1 Algorithme séquentiel d'un apprentissage de carte de Kohonen

```
pour chaque exemple faire
  pour tous les neurones de la carte de Kohonen faire
    Calcul de la distance du neurone à l'exemple                    Phase 1
  fin pour
  Détermination du neurone vainqueur : le plus proche de l'exemple  Phase 2
  pour Tous les neurones de la grille faire                         Phase 3
    si neurone vainqueur alors
      Modification des poids du neurones pour le rapprocher fortement de l'exemple
    sinon
      Modification des poids du neurone selon le principe du chapeau mexicain
    fin si
  fin pour
fin pour
```

1. Chaque neurone détermine sa distance à l'exemple présenté en entrée. Ce calcul de distance peut, par exemple, se faire par une distance simple comme celle présentée dans l'équation 9.1, mais il en existe d'autres, en fonction des propriétés recherchées de la carte.
2. Le neurone vainqueur, neurone de plus petite distance à l'exemple, est déterminé.
3. Le neurone vainqueur et les différents neurones du réseau modifient leurs poids en fonction de leur proximité au neurone vainqueur, selon une fonction en *chapeau mexicain* (voir 2.11).

$$distance = \sum_{i=0}^{n-1} |w_i - E_i|^2 \tag{9.1}$$

9.1.2 L'algorithme distribué

Une première approche pour l'implantation de ce modèle à l'aide de notre bibliothèque serait de définir la **fonction d'itération** des neurones en trois phases, comme décrit dans l'algorithme 9.2. Les exemples présentés au réseau sont déclarés comme des variables *globales* du réseau.

Algorithme 9.2 Première approche d'une fonction d'itération d'un neurone d'une carte de Kohonen

```
Lecture de l'exemple courant
Détermination de ma distance à l'exemple                    Phase 1
Détermination du neurone vainqueur                          Phase 2
Modification de mes poids selon ma proximité au neurone vainqueur   Phase 3
```

Cet algorithme ne peut pas être effectué sur un seul cycle, un neurone ayant besoin de la distance de tous les autres neurones pour déterminer le neurone *vainqueur*. Il faut donc s'assurer de la fin de la *Phase 1* de tout le réseau avant que chaque neurone puisse exécuter la *Phase 2*. L'apprentissage d'un exemple doit donc être effectué sur deux cycles, comme le montre l'algorithme 9.3. Ce qui assure au neurone le calcul de toutes les distances avant sa détermination du vainqueur.

Algorithme 9.3 Première approche d'une fonction d'itération d'un neurone d'une carte de Kohonen, apprentissage sur deux cycles

```
si cycle pair alors
    Lecture de l'exemple courant
    Détermination de ma distance à l'exemple                Phase 1
sinon
    Détermination du neurone vainqueur                      Phase 2
    Modification de mes poids selon ma proximité au neurone vainqueur   Phase 3
fin si
```

Cette méthode implique une connexion de chaque neurone à tous les autres neurones du réseau pour obtenir leur distance à l'exemple (*Phase 2*). De plus, la *Phase 2* est effectuée, **à l'identique**, par **tous** les neurones du réseau. Cette phase ne faisant appel à aucune spécificité locale, il est donc possible, comme pour l'algorithme séquentiel du réseau, de n'effectuer cette phase qu'une seule fois, pour tous les neurones du réseau.

Pour espérer une exécution efficace, il faut alors créer, comme le montre la figure 9.2, un neurone supplémentaire, effectuant cette détermination du neurone vainqueur. Ce neurone, que nous appellerons *neurone maître* est connecté à tous les neurones de la carte, ceux-ci n'étant plus connectés qu'au neurone *maître*.

L'apprentissage d'un exemple nécessite maintenant **trois cycles**. L'algorithme de la fonction d'itération des neurones de la carte devient l'algorithme 9.4, et celui du neurone *maître* l'algorithme 9.5.

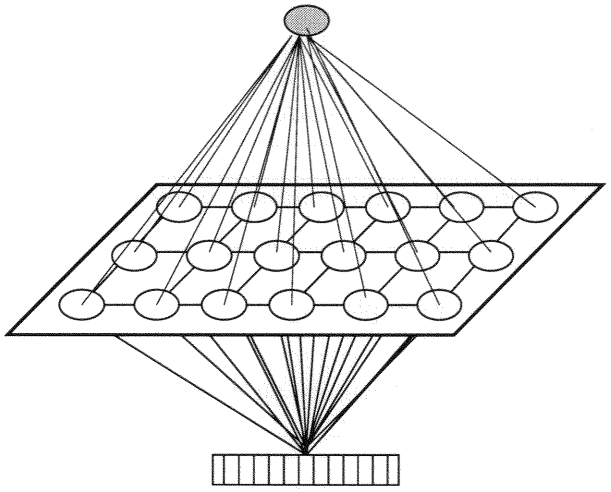


FIG. 9.2 – *Un neurone est ajouté au réseau pour effectuer le calcul du neurone vainqueur.*

Algorithme 9.4 Algorithme de la fonction d'itération des neurones de la carte sur trois cycles.

si Premier cycle **alors**
  Lecture de l'exemple courant
  Détermination de ma distance à l'exemple *Phase 1*
sinon si Second cycle **alors**
  Pas d'action *Phase 2*
sinon
  Modification de mes poids selon ma proximité au neurone vainqueur *Phase 3*
fin si

Algorithme 9.5 Algorithme de la fonction d'itération du neurone maître sur trois cycles.

si Premier cycle **alors**
  Pas d'action *Phase 1*
sinon si Second cycle **alors**
  Détermination du neurone vainqueur *Phase 2*
sinon
  Pas d'action *Phase 3*
fin si

Au cours de l'apprentissage, le rythme d'exécution de chaque neurone est donc :

	...	Nouveau cycle
	Phase 3	Nouveau cycle
Nouvel exemple	Phase 1	Nouveau cycle
	Phase 2	Nouveau cycle
	Phase 3	Nouveau cycle
Nouvel exemple	Phase 1	Nouveau cycle
	Phase 2	Nouveau cycle
	Phase 3	Nouveau cycle
	...	Nouveau cycle

En fait, et pour finir, il n'est pas indispensable de synchroniser les neurones de la carte entre les phases 3 et 1. Un neurone de la carte n'a besoin d'aucune information issue de l'exécution de la *Phase 3* des autres neurones pour calculer sa distance à un nouvel exemple (*Phase 1*).

Il est donc possible d'effectuer l'apprentissage selon le rythme suivant :

	...	Nouveau cycle
	Phase 3	Nouveau cycle
Nouvel exemple	Phase 1	
	Phase 2	Nouveau cycle
	Phase 3	Nouveau cycle
Nouvel exemple	Phase 1	
	Phase 2	Nouveau cycle
	Phase 3	Nouveau cycle
	...	

L'apprentissage d'un exemple du corpus s'effectue donc maintenant sur **deux** cycles, la lecture du premier exemple s'effectuant lors de la fonction d'initialisation des neurones de la carte. Les fonctions d'itération des neurones de la carte et du neurone maître sont alors implantées tel que montré dans les algorithmes 9.6 et 9.7.

Algorithme 9.6 Algorithme final de la fonction d'itération des neurones d'une carte de Kohonen sur deux cycles.

si Cycle pair **alors**
  Pas d'action *Phase 2*
sinon
  Modification de mes poids selon ma proximité au neurone vainqueur *Phase 3*
  Lecture du **nouvel** exemple
  Détermination de ma distance à l'exemple *Phase 1*
fin si

9.1.3 L'implantation sur le simulateur

Nous présentons maintenant les étapes les plus marquantes de l'implantation de l'algorithme défini précédemment.

Algorithme 9.7 Algorithme final de la fonction d'itération du neurone maître sur deux cycles.

si Cycle pair **alors**
  Détermination du neurone vainqueur *Phase 2*
sinon
  Pas d'action *Phases 1 et 3*
fin si

Pour distinguer le neurone maître des autres neurones de la carte nous implantons deux types de neurones, chaque type est défini par ses trois fonctions caractéristiques.

C'est le neurone *maître* qui définit, pour l'ensemble du réseau, le prochain exemple à prendre en compte. Les variables locales des différents neurones sont implantées comme montré dans l'exemple 18 pour les neurones de la carte, et dans l'exemple 19 pour le neurone *maître*. Pour permettre de transmettre toutes les données voulues, le neurone *maître* dispose d'une sortie de type structurée : `SortMait`.

```
/* Déclaration du type des variables locales des neurones */

typedef struct {
  float         distance; /*      activation du neurone       */
  float          *weight; /*         Poids du neurone         */
  canal_in         link0; /*    connexion au neurone maître   */
} NeurLocal;
```

Exemple 18 : *Type des variables locales des neurones de la carte.*

```
/* Déclaration du type de la sortie du maître */
typedef struct {
  int      exemple;       /*           Prochain exemple à traiter            */
  int    vainqueur;       /*           Dernier neurone vainqueur             */
  int    voisinage;       /* Voisinage correspondant au vainqueur ci-dessus  */
  char       etape;       /*         Etape en cours dans l'itération         */
} SortMait;


/* Déclaration du type des variables locales du neurone maître */
typedef struct {
  canal_in         *link; /* Tableau des liens avec les neurones de la carte */
  SortMait        output; /*        Sortie de type structuré du réseau       */
  int         iterations; /*              Compteur d'itérations              */
} MaitreLocal;
```

Exemple 19 : *Type des variables locales du maître.*

Pour déclarer sa sortie, le neurone *maître* doit déclarer une fonction de copie de celle-ci. Malgré le type structuré, cette fonction reste très simple, comme le montre l'exemple 20. Nous ne présenterons pas la fonction de copie des neurones de la carte, la sortie étant de type simple

`float` et la fonction de copie semblable à celle déjà vue avec l'exemple du jeu de la vie (exemple 2 page 106).

```
SortMait *CopieOutput(SortMait *modele)
{
  SortMait *tmp;

  tmp = (SortMait *) malloc(sizeof(SortMait));

  *tmp = *modele;

  return tmp;
}
```

Exemple 20 : *Fonction de copie de la sortie du neurone maître.*

Lors de la phase d'initialisation des neurones de la carte de Kohonen (exemple 21), le neurone se connecte au neurone *maître*, neurone d'immatriculation 0, et il déclare sa variable `activation` comme sortie. Lors du premier calcul de sa distance, nécessaire pour un apprentissage de chaque exemple sur deux cycles, le neurone détermine donc sa variable `activation`.

```
void init_neurone()
{
  NeurLocal *MesVarLoc;

  /*    Déclaration des variables locales    */
  MesVarLoc = (NeurLocal *) malloc(sizeof(NeurLocal));
  InitLocVar(MesVarLoc);

  /*  Initialisation des poids du neurones   */
  MesVarLoc->weight = initialisation_poids();

  /*   Déclaration de la sortie du neurone   */
  /*      ce sera juste son activation       */
  DeclareOutput(&(MesVarLoc->activation), CopieActivation);

  /*  Déclaration des connexions du neurone  */
  /*         connexion au neurone 0          */
  MesVarLoc->link0 = connect_in(0);

  /* Détermination de la premiere activation */
  MesVarLoc->activation = distance_au_modele(MesVarLoc->weight,
                                             exemple[Prems]);
}
```

Exemple 21 : *Fonction d'initialisation des neurones de la carte.*

Au cours de la fonction d'initialisation du neurone *maître*, il déclare sa variable `output` comme sortie et se connecte à tous les neurones de la carte (`nb_neurones` étant une variable globale du

programme[34]). Ensuite, le neurone *maître* détermine le prochain exemple utilisé par les neurones (le premier était, aussi, une variable locale : `Prems`, déterminée dans la partie `main()` du programme).

```
void init_master()
{
  int i;
  MaitreLocal *MesVarLoc;

  /* Déclaration des variables locales */
  MesVarLoc = (MasLocal *) malloc(sizeof(MaitreLocal));
  InitLocVar(MesVarLoc);

  MesVarLoc->output.etape      = 0;
  MesVarLoc->iteration         = 0;

  /* Déclaration de la sortie du neurone */
  DeclareOutput(&(MesVarLoc->output), CopieOutput);


  /*    Déclaration des connexions du neurone   */
  /*        Il est lié a tous les autres        */
  MesVarLoc->link = (canal_in *) malloc(nb_neurones * sizeof(canal_in));
  for (i = 0; i < nb_neurones; i ++)
    MesVarLoc->link[i] = connect_in(i+1);

  /* Détermination de la premiere activation */

  MesVarLoc->output.exemple = rand() % TAILLE;
}
```

Exemple 22 : *Fonction d'initialisation du neurone maître.*

Les exemples 23 et 24 présentent les fonctions d'itération des deux types de neurones. La répartition de l'apprentissage d'un exemple sur deux cycles est réalisée par la fonction C `switch`.

Les fonctions de terminaison (exemples 25 et 26) et la fonction `main()` du programme (exemple 27) sont données à titre d'illustration. Le tableau des exemples est une variable globale du programme, ce qui permet aux différents neurones de le lire.

34. Ce qui ne pose aucun problème car elle n'est utilisée qu'en lecture.

```
void iter_neurone()
{
  NeurLocal *MesVarLoc;
  SortMait     *Master;

  /* Récupération des variables locales */
  DecLocVar(MesVarLoc);

  /* Récupération de l'entrée */
  Master = InputCanal(SortMait, MesVarLoc->link0);

  switch (Master->etape) {
    case 0 :
      /* Le neurone maître calcule le vainqueur : rien */
     break;

    case 1 :
      /*  Mise à jour des poids des neurones */
      mise_a_jour(me(), Master->vainqueur, Master->voisinage, Master->iterations ,
                  MesVarLoc->weight);

      /* Calcul de la distance au nouvel exemple */
      MesVarLoc->ancien = Master->exemple;
      MesVarLoc->activation = distance_au_modele(MesVarLoc->weight
                                                 , exem[Master->exemple);
     break;
  }
}
```

Exemple 23 : *Fonction d'itération des neurones de la carte.*

```
void iter_master()
{
  MaitreLocal  *MesVarLoc;
  float   smallest, activ;
  int                  i;

      /*    Récupération des variables locales    */
   DecLocVar(MesVarLoc);

      /*         Condition de terminaison         */
  if (MesVarLoc->iterations > 100000)
    kill_all();
  else
    switch (MesVarLoc->etape) {
    case 0 :
      /*        Détermination du vainqueur        */
      /*        Récupération de l'entrée 0        */
      smallest = *InputCanal(float, MesVarLoc->link[0]);
      OutputField(SortMait, vainqueur) = ReturnOrigin(MesVarLoc->link[0]);
      for(i = 1; i < nb_neurones; i++) {
        /*       Récupération de l'entrée i        */
        activ = *InputCanal(int, MesVarLoc->link[i]);
        if(*activ < smallest) {
          smallest  = activ;
          OutputField(SortMait, vainqueur) = ReturnOrigin(MesVarLoc->link[i]);
        }
    }

      /*            Calcul du voisinage            */
       OutputField(SortMait, voisinage) = DetermVoisinage(MesVarLoc->iterations);

      /*    incrementation du compteur d'étapes    */
      OutputField(SortMait, etape) = 1;
     break;

    case 1 :
      /*     Détermination du prochain exemple     */
      MesVarLoc->output.exemple = rand() % TAILLE;

      MesVarLoc->iterations ++;
      OutputField(SortMait, etape) = 0;
    break;

    }
}
```

Exemple 24 : *Fonction d'itération du neurone maître.*

```
void term_neurone()
{
  NeurLocal *MesVarLoc;

  DecLocVar(MesVarLoc);
  free(MesVarLoc->weight);
  free(MesVarLoc);
}
```

Exemple 25 : *Fonction de terminaison des neurones de la carte.*

```
void term_master(void *VarLoc)
{
  MaitreLocal *MesVarLoc;

  DecLocVar(MesVarLoc);
  free(MesVarLoc->link);
  free(MesVarLoc);
}
```

Exemple 26 : *Fonction de terminaison du neurone maître.*

```
void main()
{
  int nb_process, i;

  nb_neurones = 100 * 100;                  /* Dimension de la carte de Kohonen */
  nb_process = 3;
  Prems = rand() % TAILLE;                  /* Détermination du premier exemple */
  InitNetwork(nb_neurones, nb_process);     /*          Initialisation          */
  for(i = 1; i < nb_neurones; i ++) {       /*      Création des neurones       */
    MakeNeuron(init_neurone, iter_neurone, term_neurone, i);
  }
  /*  Création du maître avec l'immatriculation "0"   */
  MakeNeuron(init_master, iter_master, term_master, 0);

  ExecuteNetwork();                         /*       Exécution du réseau        */
  FreeNetwork();                            /*           Petit ménage           */
}
```

Exemple 27 : *Fonction principale d'une carte de Kohonen.*

9.1.4 Performances

Nous présentons ici les performances obtenues avec une carte de Kohonen implantée comme présenté ci-dessus.

Ces résultats ont été obtenus sur une carte de 100 x 100 neurones, sur un apprentissage de 100000 itérations.

La figure 9.3 présente les temps d'exécution du réseau, implanté avec notre bibliothèque, en fonction du nombre de processeurs utilisés pour l'exécution.

Nous présentons, en plus des résultats obtenus, le *temps de référence séquentiel.* Ce temps correspond à l'exécution d'une même carte de Kohonen à l'aide d'un programme C séquentiel et optimisé. Le programme séquentiel est compilé avec les mêmes options que le programme de la bibliothèque et exécuté sur un processeur de *l'Origin2000.*

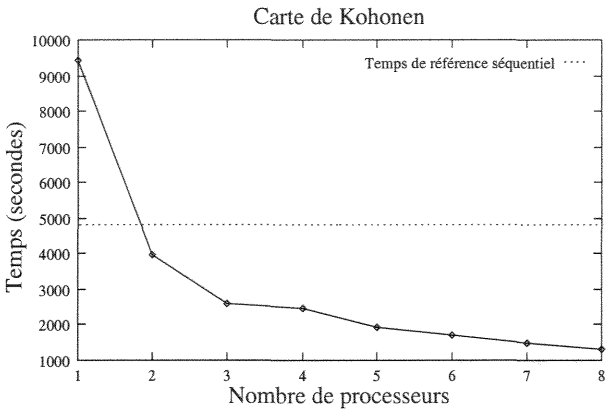


FIG. 9.3 – *Temps d'exécution d'une carte de Kohonen en fonction du nombre de processeurs utilisés. Le temps de référence séquentiel est indiqué en référence.*

La figure 9.4 représente l'accélération obtenue avec cette application, en fonction du nombre de processeurs utilisés. Cette accélération est obtenue par l'équation 9.2. A fin de comparaison, nous présentons sur cette courbe l'accélération *idéale* [35] et l'accélération par rapport au temps séquentiel de référence, obtenue par l'équation 9.3.

$$\text{Accélération sur P processeurs} = \frac{\text{Temps d'exécution sur P processeurs}}{\text{Temps d'exécution sur 1 processeur}} \tag{9.2}$$

$$\text{Accélération \% séquentiel} = \frac{\text{Temps d'exécution sur P processeurs}}{\text{Temps de référence séquentiel}} \tag{9.3}$$

Ces courbes nous donnent plusieurs informations concernant le coût séquentiel de notre bibliothèque et l'évolution temporelle de l'accélération.

Le coût séquentiel de la bibliothèque Le coût séquentiel de la bibliothèque correspond au coût, en temps d'exécution, ajouté par notre bibliothèque sur un seul processeur. Il est ici de

35. Accélération sur P processeurs = P.

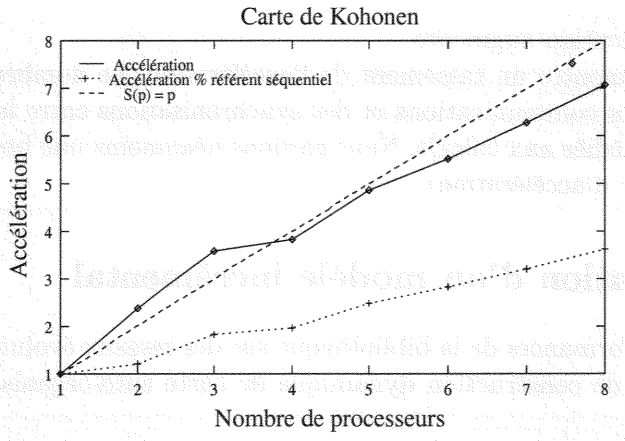


FIG. 9.4 – *Accélération d'une carte de Kohonen. Nous avons placé l'accélération idéale (Accélération = Nombre de processeurs utilisés) et l'accélération par rapport au temps d'exécution du programme séquentiel.*

1,95 : la bibliothèque effectue son exécution séquentielle 1,95 fois plus lentement que le programme séquentiel de référence. A cela plusieurs explications.

Nous avons utilisé un programme séquentiel optimisé d'apprentissage de carte de Kohonen. Ce programme, pour accélérer les applications, ne se conforme donc plus exactement aux algorithmes présentés précédemment.

Notre bibliothèque est *générale*, elle permet d'implanter toute sorte de réseaux de neurones et n'est donc pas optimisée pour un type spécifique. Par exemple, la bibliothèque exécute toutes les fonctions d'itération de tous les neurones de la carte pendant la *Phase 2*, la phase de détermination du neurone vainqueur par le neurone *maître*, le programme séquentiel ne prend pas cette peine, et cette différence se répète pour chaque itération.
L'aspect général de la bibliothèque entraîne aussi un coût en mémoire supérieur. Tout neurone de la carte de Kohonen possède les variables lui permettant potentiellement d'être, par exemple, utilisé par les fonctions spécifiques des réseaux à couches. Ce n'est évidemment pas le cas du programme séquentiel.

La bibliothèque devient toutefois plus rapide que le programme séquentiel dès l'utilisation d'un second processeur.

Les phases d'accélération Ces courbes de performance nous permettent de nous focaliser sur deux phases présentes dans l'accélération, obtenues avec peu de processeurs. Ces phases se retrouvent fréquemment sur les applications implantées avec notre simulateur. Une première phase d'hyper-accélération (de 1 à 4 processeurs utilisés) et une seconde phase de ralentissement de l'accélération (de 4 à 8 processeurs utilisés).

La première phase est essentiellement due à l'utilisation des caches mémoires. La multiplication des processeurs utilisés entraîne une augmentation de la mémoire cache, plus rapide d'accès, et un meilleur placement des données par rapport aux processeurs qui les utilisent. Ainsi, avec un seul processeur, la mémoire cache de ce processeur ne suffit pas à stocker les données qu'il utilise, d'autant que notre bibliothèque est chère en occupation de mémoire. Le nombre d'accès à des mémoires plus lentes est élevé. L'augmentation du nombre de processeurs apporte un double avantage. Chaque processeur utilise moins de données et l'espace global de mémoire cache

disponible pour l'application augmente.

La seconde phase montre un tassement de l'accélération. Le nombre de processeurs utilisés augmente et la part des communications et des synchronisations entre les processeurs augmente par rapport à la part dédiée aux calculs. Nous gardons néanmoins une bonne accélération jusqu'à huit processeurs (88 % d'accélération).

9.2 L'implantation d'un modèle incrémental

Pour tester les performances de la bibliothèque sur des réseaux évolutifs, nous avons implanté et exécuté un modèle de construction dynamique de carte auto-organisatrices : *Growing neural gas.*

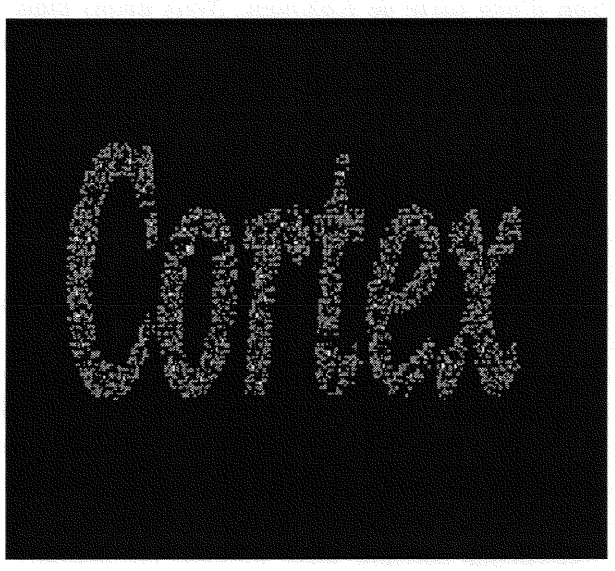


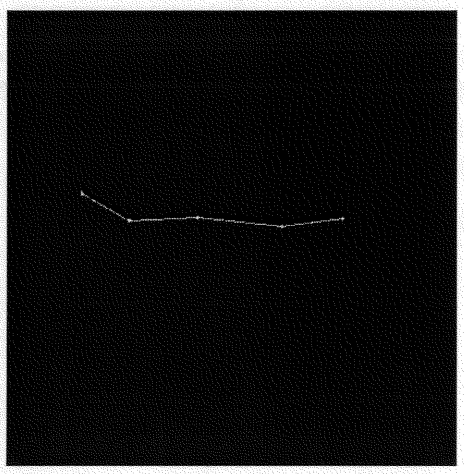
FIG. 9.5 – *Un ensemble de données...*

Comme nous l'avons précédemment vu (en page 36), un réseau de type *Growing neural gas* construit une carte auto-organisatrice pour recouvrir au mieux un espace de données. Par exemple, pour recouvrir l'espace de données représenté dans la figure 9.5, le réseau débute avec peu de neurones, deux pour notre implantation, puis ajoute des neurones au réseau au gré de l'exécution (figure 9.6), jusqu'à recouvrir de manière satisfaisante l'espace des exemples (figure 9.7).

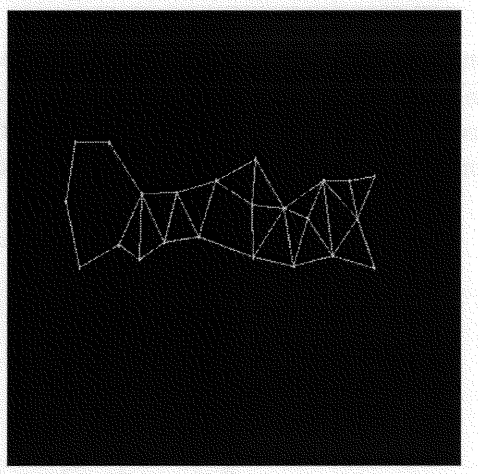
L'algorithme de *Growing neural gas* n'est pas très différent de celui des cartes de Kohonen. Les principaux changements sont la création dynamique des neurones et l'évolution par apprentissage de la topologie. Les neurones calculent leur distance de manière similaire, un neurone maître définit les deux neurones vainqueurs requis par l'algorithme. Ces deux neurones vainqueurs modifient ensuite leurs poids pour se rapprocher de l'exemple et créent ou renforcent leur connexion topologique[36].

Nous ne détaillerons ni l'algorithme ni l'implantation de cette architecture à l'aide de notre bibliothèque car ils n'apporteraient rien de plus à la compréhension des implantations. Précisons

36. Ces connexions apportent une information sur la proximité des neurones du réseau, elles ne sont pas des connexions au sens de la communication entre neurones.



(a) Le réseau démarre avec peu de neurones.



(b) Les neurones s'ajoutent pour recouvrir les entrées.

FIG. 9.6 – *Deux étapes de développement du réseau sur l'espace des entrées de la figure 9.5.*

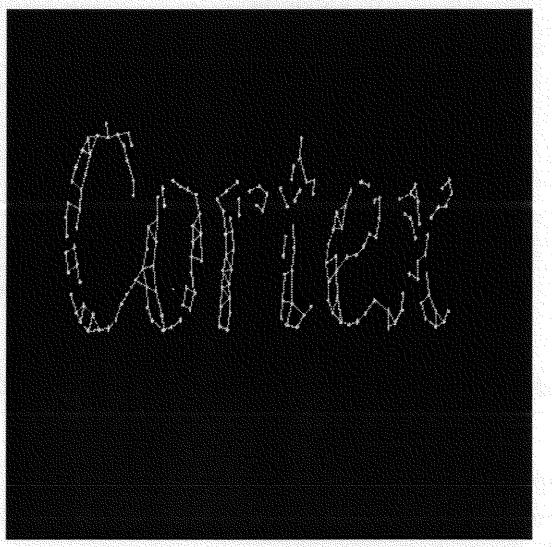


FIG. 9.7 – *Fin de l'apprentissage. Le réseau représente l'espace des données.*

seulement que le neurone *maître* gère la création et l'initialisation du neurone nouvellement créé (il est placé entre les deux neurones de plus forte erreur accumulée sur les exemples passés). Cette création se fait à l'aide de la fonction ***NewNeuron()*** fournie par la bibliothèque.

En termes de performances, les exécutions du *Growing neural gas* donnent les temps d'exécution montrés dans la figure 9.8 et les accélérations de la figure 9.9. Le réseau présenté débute l'apprentissage avec deux neurones et le termine avec 1500, le réseau est exécuté de manière totalement identique pour chaque mesure de temps.

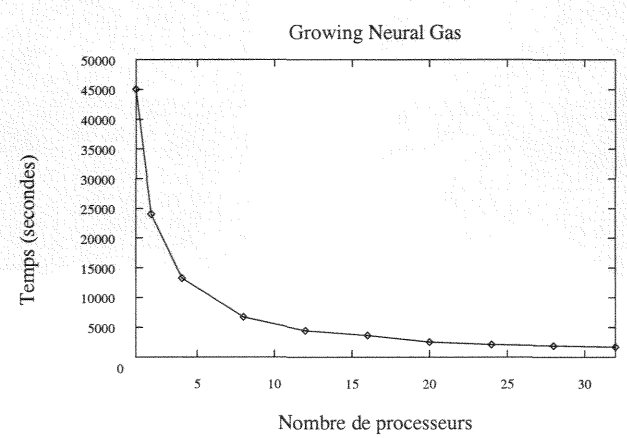


FIG. 9.8 – *Temps d'exécution d'un réseau de type* Growing neural gas *en fonction du nombre de processeurs utilisés.*

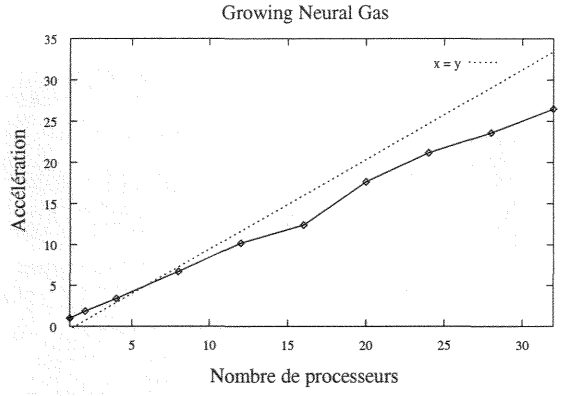


FIG. 9.9 – *Accélération du* Growing neural gas *en fonction du nombre de processeurs utilisés.*

Ces résultats nous apportent des informations plus précises sur les performances parallèles de notre bibliothèque, pour l'implantation d'architectures incrémentales et pour les exécutions sur un nombre conséquent de processeurs.

Au niveau des architectures incrémentales, l'apport de créations dynamiques de neurones n'a pas de conséquences en termes de performances. Les mises à jours des tables internes de la bibliothèque pour tenir compte des ajouts d'unités ne ralentissent pas les applications.

Les performances parallèles sur un nombre conséquent de processeurs (32 pour cette application) montrent que notre bibliothèque effectue parfaitement le transfert du parallélisme de

neurones au parallélisme matériel, nous obtenons une accélération de plus de 26 sur 32 processeurs. Comme nous l'avons vu avec l'exemple précédent, les implantations à l'aide de notre bibliothèque ne tiennent absolument pas compte des contraintes liées à la programmation sur machines parallèles : aucune des spécificités évoquées dans le chapitre 3 n'est prise en charge par le programmeur. En revanche, la bibliothèque masque bien les incompatibilités du parallélisme à grain fin des réseaux de neurones pour l'utilisation des machines MIMD. L'utilisation de la mémoire partagée pour masquer les envois de messages entre les unités des réseaux permet à la bibliothèque de réduire au mieux les coûts liés à ces échanges de données et aux fréquentes synchronisations entre les processeurs.

Les résultats, en termes de performances parallèles, sont donc satisfaisants pour des architectures connexionnistes répondant à notre formalisme, *le parallélisme de neurones*. Ce formalisme permet au simulateur d'effectuer au mieux la transition sur les machines parallèles.

9.3 L'implantation d'un modèle à couches

Nous présentons maintenant un modèle connexionniste développé dans notre équipe. Nous avons implanté l'élagage d'un réseau contextuel. Ce réseau contextuel est un perceptron multi-couches pour lequel chaque poids est déterminé par un réseau connexionniste propre, dit *OWE*, comme montré dans la figure 9.10. Les réseaux OWE sont des perceptrons multi-couches, recevant les mêmes entrées que le réseau principal. Notre équipe utilise cette architecture pour effectuer de la *prédiction d'atténuation de champs radioélectrique* [Bougrain, 2000] dans le cadre d'une collaboration avec France Télécom. Il s'agit de ce que l'on peut appeler une grosse application, tant par le nombre d'entrées du réseau (32) que par la taille des corpus qui se chiffre en dizaines de milliers d'exemples.

Nous utilisons un réseau contenant trois couches, une couche d'entrée de 32 neurones, une couche cachée de 10 neurones et un unique neurone en sortie. Les OWE, réseaux de déterminations des poids du réseau principal, disposent de la même topologie ($32 \times 10 \times 1$). Nous obtenons ainsi un réseau à 14190 neurones, pour une architecture contenant 993 couches.

L'élagage de ce réseau a pour objectif de mettre en valeur les paramètres importants de l'espace des entrées et de transformer un réseau "lourd" en un réseau plus facilement maniable. A titre d'illustration, la figure 9.11 présente l'état du réseau de la figure 9.10 après élagage. Il semble intéressant d'implanter ce réseau à l'aide de notre bibliothèque. Cela permet de disposer de la puissance du parallélisme pour l'apprentissage et d'exécuter ensuite le réseau élagué, moins lourd en calcul, sur plate-forme séquentielle.

La phase d'apprentissage de ce réseau, phase qui comprend l'élagage, est basée sur un algorithme de rétro-propagation de l'erreur.

Les courbes 9.12 et 9.13 présentent les performances obtenues pour la phase d'apprentissage de ce réseau.

Les performances obtenues sur cette architecture sont peu satisfaisantes : l'apprentissage est trois fois plus rapide sur huit à dix processeurs (voir figure 9.13). Ces résultats s'expliquent par la philosophie même de notre bibliothèque. Ce simulateur propose d'utiliser le parallélisme *intrinsèque* des modèles connexionnistes pour l'implantation sur machine parallèle. Or l'algorithme de rétro-propagation du gradient ne correspond pas à ce parallélisme.

Cet algorithme peut être considéré comme *séquentiel par couche*. En effet, les couches sont

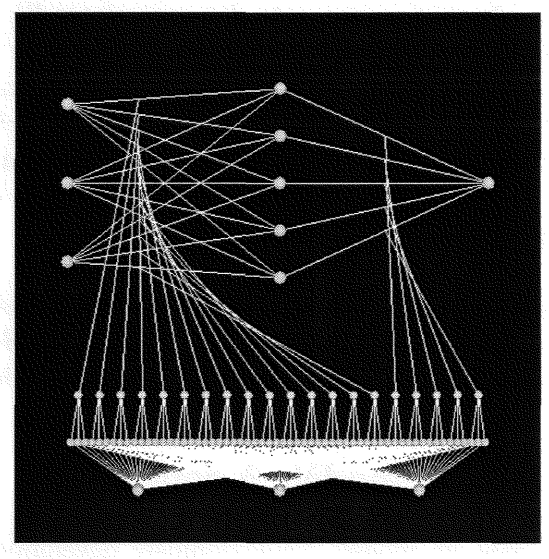


FIG. 9.10 – *Un réseau à couche prenant en compte le contexte dans le calcul de ses poids. Chaque poids des connexions du réseau principal (représenté en haut, horizontalement) est déterminé par un perceptron multi-couches, appelé OWE (représenté en bas, verticalement). Les entrées du réseau principal (trois sur la figure) sont identiques aux entrées présentées aux réseaux OWE.*

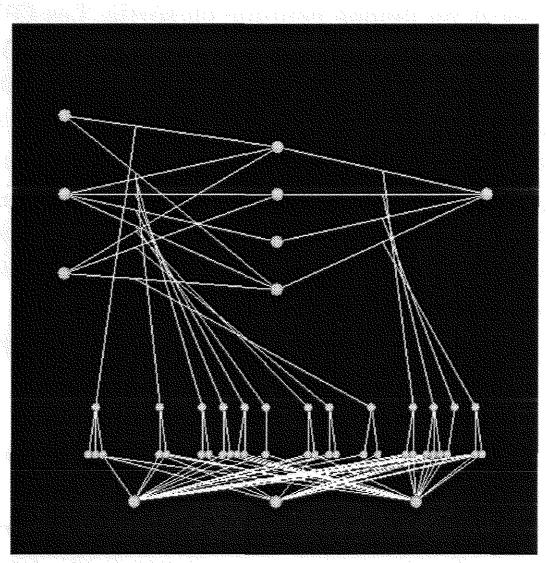


FIG. 9.11 – *Réseau contextuel après élagage.*

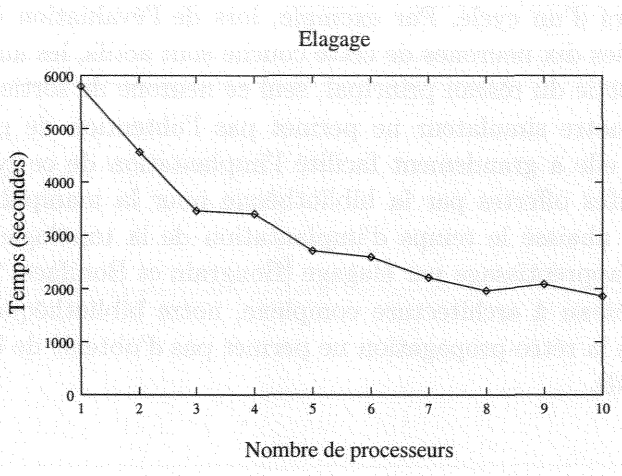


FIG. 9.12 – *Temps d'apprentissage du réseau contextuel en fonction du nombre de processeurs utilisés.*

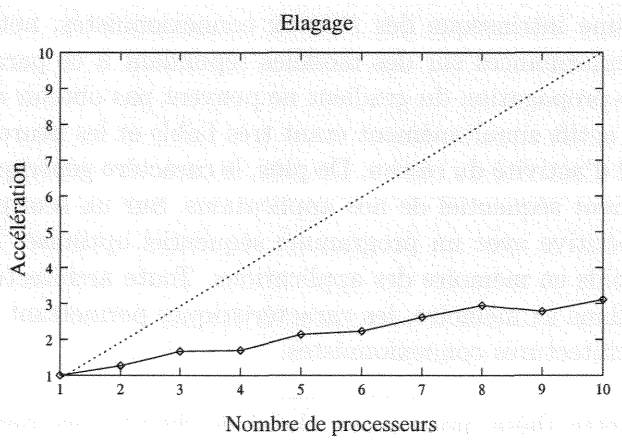


FIG. 9.13 – *Accélération de l'apprentissage du réseau contextuel en fonction du nombre de processeurs utilisés.*

évaluées les unes à la suite des autres, des entrées vers la sortie, pour calculer l'activité du réseau, puis de la sortie vers l'entrée pour rétro-propager l'erreur, et cela pour chaque exemple du corpus. En fait, et malgré le nombre de neurones présents dans le réseau (14190), très peu sont actifs simultanément au cours d'un cycle. Par exemple, lors de l'évaluation de la couche cachée du réseau principal, seuls les dix neurones de cette couche sont actifs, les autres étant inactifs. Lors de l'évaluation de la sortie du réseau principal, seul ce neurone de sortie est actif.

Si l'utilisation de notre simulateur ne permet pas l'obtention de performances en termes de temps d'exécution, elle a grandement facilité l'implantation de ce type d'architectures. Les différentes fonctionnalités offertes par la bibliothèque pour la manipulation des neurones par couches ont fortement abaissé le temps d'implantation de la topologie du réseau et le temps d'implantation de son apprentissage par élagage [Bougrain et Boniface, 2000; Bougrain, 2000].

Pour ce type de réseau à architecture complexe, notre bibliothèque permet donc une implantation rapide, mais la rétro-propagation ne permet pas d'obtenir de bonnes performances en temps effectifs de calculs.

9.4 Conclusion

Les exemples d'implantation vus dans ce chapitre nous éclairent sur plusieurs caractéristiques de la bibliothèque.

En termes d'implantation, la bibliothèque permet d'implanter toute sorte de réseaux de neurones. Une simulation effectuée à l'aide de la bibliothèque n'utilise que le parallélisme des réseaux de neurones pour l'implantation. Aucun point de l'implantation n'est dicté par les contraintes de programmation sur machine parallèle. L'algorithme séquentiel, ramené au niveau des neurones, peut donc être utilisé pour effectuer ces implantations.

En termes de temps d'exécution parallèle, si notre bibliothèque permet d'implanter des réseaux d'architectures variées, les performances obtenues sont plus hétérogènes. Utilisant les propriétés de parallélisme intrinsèque des réseaux connexionnistes, notre bibliothèque permet d'obtenir de bonnes performances sur des modèles répondant à ce parallélisme. Les réseaux à apprentissage par rétro-propagation du gradient ne peuvent pas obtenir de bonnes accélérations, le nombre de neurones actifs simultanément étant très faible et les neurones étant peu de temps actifs sur le temps total d'activité du réseau. De plus, le caractère générique de notre bibliothèque entraîne un ralentissement séquentiel de nos applications. Sur un seul processeur, notre bibliothèque n'est pas compétitive avec un programme séquentiel optimisé. L'une des raisons de ce ralentissement est le poids en mémoire des applications. Toute architecture développée porte en elle, plus précisément dans sa mémoire, les caractéristiques permettant l'écriture et l'exécution de toutes les autres architectures connexionnistes.

Dans le cadre de cette thèse, nous avons choisi de discuter les performances de notre bibliothèque sur des modèles classiques ou utilisant des algorithmes classiques bien diffusés dans la communauté, en négligeant les modèles d'inspiration biologique de diffusion (encore) restreinte. Ce choix nous permet de présenter les algorithmes (simples) de ces modèles et leur implantation, alors que la présentation des algorithmes, plus complexes [Frezza-Buet, 1999; Rougier, 2000], des modèles biologiques aurait certainement été plus fastidieuse pour le lecteur. Il est cependant clair que cette bibliothèque a aussi (et peut-être surtout) pour vocation d'implanter des modèles biologiques tels que ceux développés dans notre équipe. Si de telles réalisations sont encore en cours d'étude, les résultats obtenus avec les réseaux dynamiques *Growing*

Neural Gas laissent espérer de bons résultats, tant pour les facilités d'implantation que pour les performances parallèles.

Nous n'avons présenté ici que des exemples de réseaux de grande taille, la recherche de puissance de calcul n'étant effective que pour ce type de réseaux. Mais il est clair que, pour obtenir des résultats significatifs en performances sur machine parallèle, notre bibliothèque nécessite des réseaux comportant un grand nombre de neurones. Il faut en effet un nombre de neurones suffisant pour que les temps de calcul des neurones au cours d'un cycle recouvrent les temps de migration des différentes zones mémoires partagées et les phases de synchronisation.

Conclusion

Nous avons présenté dans ce manuscrit un nouveau simulateur de réseaux de neurones artificiels. Ce simulateur, présenté sous la forme d'une bibliothèque de fonctions sur le langage C, propose de développer des réseaux connexionnistes en s'appuyant sur un formalisme proche du modèle biologique, *le parallélisme de neurones*. Il permet d'implanter des réseaux de grande taille et d'utiliser les machines parallèles de type *MIMD à mémoire partagée* pour les exécutions.

Une étude détaillée des modèles connexionnistes nous a permis de mettre à jour les grandes caractéristiques, parallèles et topologiques, des réseaux de neurones. De cette étude, nous avons extrait le formalisme parallèle des modèles proposés pour leur implantation, puis les besoins en termes de construction des réseaux, d'implantation des unités de base des réseaux et des connexions entre ces unités.

De l'étude du parallélisme informatique, nous avons extrait les spécificités liées à la programmation sur les machines parallèles actuelles, puis les aspects technologiques importants de ces machines.

Ces deux études ont convergé pour aboutir à notre bibliothèque de simulation de réseaux de neurones. Pour proposer un outil attractif et efficace, cette bibliothèque est une passerelle entre deux parallélismes distincts. Les réseaux de neurones sont développés suivant un parallélisme à grain fin, avec un protocole de communication entre les unités, protocole proche de la communication par envois de messages. Les exécutions parallèles répondent à un parallélisme à gros grain, utilisant un protocole de partage de mémoire pour les communications entre les différents processeurs. Le point central de nos travaux est l'utilisation des spécificités parallèles des modèles connexionnistes pour permettre l'exploitation du parallélisme matériel, le premier parallélisme *masquant* totalement le second.

C'est l'utilisation de la mémoire partagée, à notre connaissance inédite dans le domaine de la simulation des réseaux connexionnistes, qui permet d'obtenir de bons résultats sur nos deux objectifs principaux en matière de simulateur :

- Les programmes s'implantent en ramenant les algorithmes séquentiels classiques au niveau du parallélisme naturel des neurones, sans interférences du parallélisme matériel. L'implantation des réseaux au niveau des neurones facilite l'écriture, la lisibilité et la modification des programmes. Ainsi, en utilisant notre simulateur et les fonctions qu'il propose, un utilisateur pense et implante ses réseaux comme un ensemble de neurones pouvant être exécutés de manière concurrente. La bibliothèque offre les fonctions permettant au programmeur de définir des neurones et leurs connexions.
- Les performances sur machines parallèles MIMD à mémoire partagée sont satisfaisantes pour les réseaux répondant à notre modèle théorique, c'est-à-dire les réseaux dont les neurones sont actifs simultanément au cours de l'exécution du réseau.

Néanmoins notre bibliothèque accompagne sa généricité d'un coût séquentiel conséquent et

d'un fort coût en espace mémoire pour les applications. Si nos applications peuvent être exécutées sur machines séquentielles, elles n'y sont pas en concurrence avec des implantations optimisées. C'est avec les exécutions sur machines parallèles, sur lesquelles nos applications sont implantées pour être performantes que la comparaison doit se faire. De plus, les performances obtenues sont limitées aux réseaux répondant à notre formalisme. Notre simulateur ne peut pas laisser espérer de performances sur des algorithmes ne répondant pas au *parallélisme de neurones*, comme la rétro-propagation du gradient.

Une propriété importante de notre simulateur est qu'il permet l'implantation de réseaux de grandes tailles. L'utilisation des machines parallèles permet de disposer d'une importante puissance de calcul et d'espace mémoire conséquent. Le formalisme d'implantation des réseaux permet de plus le développement de grands nombres d'unités, connectées selon des topologies variées, tout en gardant des codes lisibles. L'implantation des neurones par la définition des fonctions caractéristiques de leur type, à l'instar de la programmation objet, laisse le code clair, quel que soit le nombre de types différents de neurones implantés (il est aussi aisé d'ajouter, de soustraire ou de modifier complètement un type de neurones sans rien modifier au reste du réseau).

C'est pour cette facilité d'implantation que nous avons décidé d'utiliser ce simulateur pour développer le modèle de réseau à couche contextuel. Le simulateur nous a ainsi permis de développer le réseau, à partir de l'algorithme de base, et d'ajouter l'élagage à son apprentissage sans grandes difficultés.

Pour toutes ces raisons, un objectif important de notre travail serait atteint si notre simulateur devenait l'outil de développement de référence pour les modèles d'inspiration biologique développés au sein de notre équipe. Ces réseaux sont en effet connus pour la complexité de leurs implantations et pour leur coût en temps de calcul, coût dû au nombre d'unités en constante croissance.

Le formalisme proposé par le simulateur, associé à la possibilité de travailler avec un nombre élevé de neurones, nous permet d'envisager l'étude des caractères et propriétés émergeant des populations de neurones, et même de construire nos modèles en s'appuyant sur ces populations plutôt que de les simuler.

En termes de perspectives, notre bibliothèque semble au début de son histoire. Outre les améliorations que nous souhaitons apporter au modèle de simulateur présenté, nous travaillons actuellement au développement de la bibliothèque, en termes de spécificités techniques comme en termes de diffusion auprès de la communauté scientifique.

La première évolution technique consistera à baisser le coût séquentiel de notre bibliothèque. Nous avons pour cela plusieurs pistes. Il est possible d'abaisser le poids des réseaux en mémoire en abaissant le poids de chacun des neurones. Comme nous l'avons vu, tout neurone possède potentiellement toutes les fonctionnalités offertes par la bibliothèque, par exemple la gestion par couches. Le choix de développement d'une bibliothèque nous interdit l'interprétation des réseaux avant exécution, interprétation qui aurait permis de spécialiser les neurones. Il est possible de pallier cette difficulté en favorisant le dialogue entre l'utilisateur et la bibliothèque par l'intermédiaire des options de compilation. La multiplication de ces options de compilation permettrait de spécialiser les neurones sur les options topologiques choisies par l'utilisateur, tout en lui laissant la possibilité de les croiser afin de développer toute sorte d'architecture exotique.

En termes d'enrichissement des outils proposés par la bibliothèque, quelques fonctions pouvant apporter un confort de programmation peuvent être envisagées. Il serait par exemple possible

de proposer une fonction permettant de spécifier à la bibliothèque qu'un neurone souhaite être inactif pendant un nombre fini de cycles.

Si des fonctions peuvent être ajoutées, il est aussi possible d'améliorer le simulateur à d'autres niveaux. Nous travaillons actuellement à la mise au point de macros permettant de spécifier différents types d'unités. Nous spécifions en particulier des modèles de colonnes corticales, de maxi-colonnes et, à une autre granularité, de neurones à *spikes*.

Pour faciliter les développements et les études des réseaux (mais aussi les indispensables démonstrations logicielles ...), nous finalisons actuellement un ensemble de fonctions graphiques adaptées à la bibliothèque et à son formalisme. Le formalisme proposé par la bibliothèque entraîne la création et la programmation explicite des différents neurones et de leurs connexions. La nature même de ce formalisme rend intéressant de proposer un outil graphique permettant de visualiser les neurones et les connexions constituant un réseau. Ces visualisations peuvent être très utiles pour les phases d'expérimentation des réseaux, notamment pour suivre leur évolution topologique. Ces fonctions permettent les visualisations présentées dans les figures 9.5, 9.6, 9.7, 9.10 et 9.11, et permettront d'obtenir de nombreuses informations sur les neurones en cours d'exécution du réseau. Pour ne pas perdre l'apport du parallélisme, les fonctions graphiques sont gérées par un *pipe graphique* sur l'*Origin2000*.

Nous souhaitons enfin diffuser notre simulateur et le présenter à d'autres équipes connexionnistes, pour leur en proposer l'utilisation et obtenir des retours de ces nouveaux utilisateurs. Une collaboration avec l'université d'Amsterdam est en cours, dans ce sens.

Ecrite pour être implantée sur ordinateurs parallèles de type MIMD à mémoire partagée, notre bibliothèque devrait pouvoir résister à quelques générations d'ordinateurs parallèles, les grands constructeurs de cette technologie misant actuellement sur OpenMP, protocole de programmation adapté à ces architectures parallèles.

S'il est peu probable de voir émerger une technologie inadéquate pour les architectures des machines parallèles, il est actuellement possible d'envisager une augmentation de l'offre en matière de *cluster de PCs*. Ces réseaux, à hauts débits, d'ordinateurs séquentiels ont les avantages d'un coût financier nettement inférieur aux machines parallèles et d'un renouvellement de puissance potentiellement fréquent. Actuellement, ces réseaux fonctionnent sur la base d'envois de messages entre les différents modules du réseau.

Notre simulateur ne peut pas être utilisé directement sur ce type d'architecture parallèle. Néanmoins, les recherches en cours du côté des protocoles de *mémoire virtuellement partagée* laissent envisager des potentialités d'adaptation de notre bibliothèque sur ces réseaux, mais avec des performances probablement moindres. De plus, au vu de l'évolution technologique, il est probable que des communications par partage de mémoire entre les différents modules de ces réseaux seront "rapidement disponibles" et efficaces sur ces architectures.

Le gros problème restant à résoudre, rapidement, est de trouver un nom à ce simulateur de réseaux de neurones artificiels implanté sous la forme d'une bibliothèque de fonctions dédiées au connexionnisme, ce nom aurait en effet permis d'alléger agréablement la lecture de ce manuscrit, à commencer par cette phrase.

Bibliographie

[Abdi, 1994] Hervé Abdi. *Les Réseaux de Neurones (French) [Neural Networks].* Presses Universitaires de Grenoble, Grenoble, France, 1994.

[Alexandre et Guyot, 1995] F. Alexandre et F. Guyot. Neurobiological Inspiration for the Architecture and Functioning of Cooperating Neural Networks. Dans *Proceedings International Workshop on Artificial Neural Networks*, Malaga (Spain), June 1995.

[Alexandre, 1990] Frédéric Alexandre. *Une modélisation fonctionnelle du cortex : la colonne corticale.* PhD thesis, U.H.P. Nancy I, 1990.

[Alpaydin, 1990] E. Alpaydin. *Neural models of incremental supervised and unsupervised learning.* PhD thesis, Ecole Polytechnique Fédérale Lausanne, 1990. Thèse No. 863(1990).

[Amdhal, 1967] G. M Amdhal. Validity of the single-processor approach to achieving large-scale computing capabilities. Dans *AFIPS Conference Proceedings*, pages 483–485. AFIPS Press, 1967.

[Amit *et al.*, 1985] D. Amit, H. Gutfreund et H. Sompolinsky. Spin-glass models of neural networks. *Physical Review*, A 32:1007–1018, 1985.

[Arbib, 1995] M. A. Arbib. *Handbook of Brain Theory and Neural Networks.* MIT Press, 1995.

[Barbosa et Lima, 1990] Valmir C. Barbosa et Priscilla M. V. Lima. On the distributed parallel simulation of Hopfield's neural networks. *Software, Practice and Experience*, 20(10):967–983, Octobre 1990.

[Blank, 1990] T. Blank. The MASPAR MP-1 architecture. Dans *Thirty-Fifth IEEE Computer Society International Conference - Compcon Spring 90*, pages 20–4, San Francisco, CA, 1990.

[Boniface, 1994] Yann Boniface. Prédiction de structure secondaire d'ARN sur une machine parallèle. Mémoire de DEA, Université de Rouen, juin 1994.

[Boser *et al.*, 1992] Bernhard E. Boser, Isabelle M. Guyon et Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. Dans *Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory*, rédacteur David Haussler, pages 144–152, Pittsburgh, PA, Juillet 1992. ACM Press.

[Bougrain et Boniface, 2000] Laurent Bougrain et Yann Boniface. Détermination de l'architecture de modèles neuromimétiques par implantation parallèle. Dans *Journées Scientifiques du Centre Charles Hermite*, Janvier 2000.

[Bougrain, 2000] L Bougrain. *Étude de la construction par réseaux neuromimétiques de représentations interprétables : Application à la prédiction dans le domaine des télécommunications.* PhD thesis, Université Nancy 1, 2000.

[Briggs et Dubois, 1986] F. A. Briggs et M. Dubois. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. Dans *Proc. of the International Conference on Parallel Processing*, 1986.

[Burkhardt *et al.*, 1992] H. Burkhardt, S. Frank, B. Knobe et J. Rothnie. Overview of the KSR1 computer system. Rapport Technique KSR-TR-9202001, Kendall Square Research, Février 1992.

[Burnod, 1989] Yves Burnod. *An adaptive neural network : the cerebral cortex.* Masson, 1989.

[Chaiken, 1990] D. L. Chaiken. Cache Coherence Protocol for large-scale multiprocessors. Technical Report MIT/LCS/TR-489, Massachusetts Institute of Technology, Septembre 1990.

[Chase *et al.*, 1989] Jeffery S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy et Richard J. Littlefield. The amber system: parallel programming on a network of multiprocessors. Dans *Proceedings of the 12th ACM Symposium on Operating System Principles*, volume 23, pages 147–58, Décembre 1989.

[Chauvin, 1990] Y. Chauvin. Generalization performance of overtrained backpropagation networks. Dans *EURAPSIP Workshop*, rédacteurs L.Almeida et C.Wellekens, Sesimbra(Portugal), 1990. Springer-Verlag. p46-55.

[Chester, 1990] D. L. Chester. Why two hidden layers are better than one. Dans *Proceedings of the International Joint Conference on Neural Networks*, volume 1, pages 265–268. Erlbaum, 1990.

[Chu et Wah, 1992] Lon-Chan Chu et Benjamin W. Wah. Optimal mapping of neural-network learning on message-passing multicomputers. *Journal of Parallel and Distributed Computing*, 14(3):319–339, Mars 1992.

[Clary et Kothari, 1991] J. S. Clary et S. Kothari. Implementation of the hopfield neural network on the maspar system. Dans *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, Architecture, pages I–676–I–677, Boca Raton, FL, Août 1991. CRC Press.

[Cornu, 1992] T. Cornu. *Machine cellulaire virtuelle définition, implantation et exploitation.* PhD thesis, Université Nancy 1, Nancy, octobre 1992.

[Cortes et Vapnik, 1995] Corinna Cortes et Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[Coulaud et Dillon, 1996] Olivier Coulaud et Eric Dillon. Early Implementation of Para++ with MPI-2. Dans *Second MPI Developer's Conference - MDC'96, Notre-Dame*, Juillet 1996.

[Cowan, 1989] Jack D. Cowan. Neural networks: The early days. Dans *[Touretzky, 1990]*, pages 828–842, 1989.

[Crespo et Mora, 1993] J. L. Crespo et E. Mora. Tests of different regularization terms in small networks. *Lecture Notes in Computer Science*, 686:p 284, 1993.

[Dagum et Menon, 1998] L Dagum et R Menon. Openmp : an industry-standard api for shared-memory programming. *IEEE Computational science and engineering*, january-march 1998.

[Demian et Mignot, 1996] V. Demian et J. C. Mignot. Implementation of the self-organizing feature map on parallel computers. *Computers and Artificial Intelligence*, 15(1):63–80, 1996.

[Demuth et Beale, 1993] Howard Demuth et Mark Beale. *Neural Network Toolbox: For use with MATLAB: User's Guide.* The Mathworks, Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1993.

[Durand, 1995] Stephane Durand. *TOM, une architecture connexionniste de traitement de séquences. Application à la reconnaissance de la parole.* PhD thesis, Université Henri Poincaré, Nancy I, 1995.

[Elman, 1990] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[Fahlman et Lebiere, 1990] S. E. Fahlman et C. Lebiere. The cascade-correlation learning architecture. Dans *Advances in Neural Information Processing Systems 2*, rédacteur D. S. Touretzky. Morgen Kaufmann, 1990.

[Fahlman, 1989] S. E. Fahlman. Fast-learning variations on back-propagation: An empirical study. Dans *Proceedings of the 1988 Connectionist Models Summer School*, rédacteurs D. Touretzky, G. Hinton et T. Sejnowski, pages 38–51, San Mateo, 1989. (Pittsburg 1988), Morgan Kaufmann.

[Fahlman, 1991] Scott E. Fahlman. The recurrent cascade-correlation architecture. Dans *Advances in Neural Information Processing Systems*, rédacteurs Richard P. Lippmann, John E. Moody et David S. Touretzky, volume 3, pages 190–196. Morgan Kaufmann Publishers, Inc., 1991.

[Fort, 1988] J. C. Fort. Solving a combinatorial problem via self-organizing process: an application of the Kohonen algorithm to the Traveling Salesman Problem. *Biol. Cyb.*, 59(1):33–40, 1988.

[Frezza-Buet *et al.*, 2000] H. Frezza-Buet, N. Rougier et F. Alexandre. *Sequence Learning: Paradigms, Algorithms and Applications*, chapitre Integration of biologically inspired temporal mechanisms into a cortical framework for sequence processing. Giles, L. and Sun, R., 2000.

[Frezza-Buet, 1999] H. Frezza-Buet. *un modèle de cortex pour le comportement motivé d'un agent neuromimétique autonome.* PhD thesis, Université Henri Poincaré, nancy I, 1999.

[Fritzke, 1995] Bernd Fritzke. A growing neural gas network learns topologies. Dans *Advances in Neural Information Processing Systems*, rédacteurs G. Tesauro, D. Touretzky et T. Leen, volume 7, pages 625–632. The MIT Press, 1995.

[Gas, 1994] B. Gas. *Un modèle connexionniste non supervisé pour l'apprentissage et la reconnaissance de séquences temporelles.* PhD thesis, Université Paris Sud, Orsay, 1994.

[Gégout *et al.*, 1995] C. Gégout, B. Girau et F. Rossi. A general feedforward neural network model. Research report NC-TR-95-041, 1995.

[Geist, 1994] Al Geist. *PVM, a users' guide and tutorial for networked parallel computing.* Mit Press, 1994.

[Gerstner, 1998] W. Gerstner. Spiking neurons. Dans *Pulsed Neural Networks*, rédacteurs W. Maass et C. Bishop. Bradford Book, MIT Press, 1998.

[Girard et Paugam-Moisy, 1994] D. Girard et H. Paugam-Moisy. Strategies of weight updating for parallel back-propagation. Dans *Applications in Parallel and Distributed Computing*, rédacteur C. Girault, volume A-44, pages 335–336, North-Holland, 1994. IFIP Transactions.

[Girau et Paugam-Moisy, 1995] B. Girau et H. Paugam-Moisy. Load sharing in the training set partition algorithm for parallel neural learning. Dans *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95*, pages 586–591, Los Alamitos, CA, USA, Avril 1995. IEEE Computer Society Press.

[Girau, 1995] B. Girau. Mapping neural network back-propagation onto parallel computers with computation/communication overlapping. Dans *Proceedings of the First International EURO-PAR Conference*, rédacteurs Seif Haridi, Khayri Ali et Peter Magnusson, Lecture Notes in Computer Science, pages 513–524, Stockholm, Sweden, Août 29–31, 1995. Springer-Verlag.

[Girau, 1999] Bernard Girau. *Du parallélisme des modèles connexionnistes à leur implantation parallèle.* Thèse, ENS Lyon, Mars 1999.

[Goddard *et al.*, 1989] N. Goddard, K. Lynne, T. Mintz et L. Bukys. Rochester connectionist simulator. Rapport Technique TR-233 (revised), Computer Science Dept, University of Rochester, 1989.

[Gropp *et al.*, 1994] William Gropp, Ewing Lusk et Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambrdge, MA, Octobre 1994.

[Gropp, 1999] William Gropp. *Using MPI-2.* Mit Press, 1999.

[Grossberg, 1984] Stephen Grossberg. Some normal and abnormal behavioral syndromes due to transmitter gating of opponent processes. *Biological Psychiatry*, 19(7):1075–1117, 1984.

[Guigon, 1993] Emmanuel Guigon. *Modélisation des propriétés du cortex cérébral : Comparaison entre aires visuelles, motrices et préfrontales.* PhD thesis, École centrale de Paris, 1993. in English.

[Guyot, 1990] F. Guyot. *Une modélisation fonctionnelle du cortex : la Colonne Corticale. Aspects auditifs et visuels.* PhD thesis, Université Henri Poincaré, Nancy I, Vandoeuvre-lès-Nancy, Avril 1990.

[Hassibi et Stork, 1993] B. Hassibi et D. G. Stork. Second derivatives for network pruning: Optimal Brain Surgeon. Dans *Advances in Neural Information Processing Systems 5. Proceedings of the 1992 Conference*, rédacteurs S. J. Hanson, J. D. Cowan et C. L. Giles, pages 164–171, San Mateo, CA, 1993. Morgan Kaufmann.

[Hebb, 1949] D. O. Hebb. *The Organization of Behavior.* Wiley & Sons, New York, 1949. *Hebb's "cell-assembly" model, in which aggregates of neurons form the physiological basis of concepts.*

[Hecht-Nielsen, 1989] R. Hecht-Nielsen. Theory of the backpropagation neural network. Dans *International Joint Conference on Neural Networks*, volume 1, pages 593–605, New York, 1989. (Washington 1989), IEEE.

[Hecht-Nielsen, 1990] Robert Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, Reading, MA, USA, 1990.

[Hertz *et al.*, 1991] John Hertz, Anders Krogh et Richard G. Palmer. *An Introduction to the Theory of Neural Computation.* Lecture Notes Volume I. Addison Wesley, 1991.

[Hillis, 1985] D. W. Hillis. *The Connection Machine.* MIT Press, Cambridge, Mass., 1 édition, 1985.

[Hinton et Sejnowski, 1986] Geoffrey E. Hinton et T. J. Sejnowski. Learning and relearning in boltzmann machines. Dans *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, rédacteur Rumelhart D. E., Cambridge, MA, 1986. Bradford Books.

[Hoehfeld et Fahlman, 1992] Markus Hoehfeld et Scott E. Fahlman. Learning with limited numerical precision using the Cascade-Correlation algorithm. *IEEE Transactions on Neural Networks*, 3(4):602–611, July 1992.

[Hopfield, 1982] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences (USA)*, 79:2554–2558, 1982.

[Hopp et Prechelt, 1997] Holger Hopp et Lutz Prechelt. Cupit-2 - a parallel language for neural algorithms: language reference and tutorial. Technical Report iratr-1997-4, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, 1997.

[Hopp et Prechelt, 1999] Holger Hopp et Lutz Prechelt. Cupit-2: Portable and efficient high-level parallel programming of neural networks. *Systems Analysis, Modelling, Simulation (SAMS)*, 34(4), 1999.

[Hornik *et al.*, 1989] Kurt Hornik, M. Stinchcombe et H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[Hérault et Jutten, 1994] Jeanny Hérault et Christian Jutten. *Réseaux neuronaux et traitement du signal.* Hermes, 1994.

[Hubel et Wiesel, 1977] D. H. Hubel et T. N. Wiesel. Functional architecture of macaque monkey visual cortex. *Ferrier Lecture Proc. Roy. Soc. London*, pages 1–59, 1977.

[Jodouin, 1994] Jean-Francois Jodouin. *Les Réseaux de neurones, principes et définitions.* Hermes, 1994.

[Jordan, 1986] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. Dans *Proceedings of the Eighth Annual Conference of the Cognitive Science Society.* Erlbaum, 1986.

[Jul *et al.*, 1988] Eric Jul, Henry Levy, Norman Hutchinson et Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Février 1988.

[Karnin, 1990] Ehud D. Karnin. A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242, 1990.

[Kermarrec et Morin, 1997] A.-M. Kermarrec et C. Morin. Icare : une mémoire virtuelle partagée alliant tolérance aux fautes et efficacité dans un réseau de stations de travail. *Technique et Science Informatiques*, 16(10):1257–1282, 1997.

[Kermarrec, 1996] A.-M. Kermarrec. *Une approche globale fondée sur la réplication pour la disponibilité et l'efficacité des systèmes extensibles à mémoire partagéee.* PhD thesis, IRISA, 1996.

[Kock *et al.*, 1996] G. Kock, M. Endler, M.D. Gubitosi et S.W. Song. Towards transparent parallelization of connectionist systems. Dans *Ninth Int. Conf. on Parallel and Distributed Computing Systems (PDCS'96)*, Dijon(France), 1996.

[Kock et Serbedzija, 1994] G. Kock et N.B. Serbedzija. Artificial neural networks: From compact descriptions to C++. Dans *ICANN'94*, 1994.

[Kock et Serbedzija, 1996] G. Kock et N.B. Serbedzija. Simulations of artificial neural networks. *Systems Analysis - Modelling - Simulation (SAMS)*, 27(1):15–59, 1996.

[Kohonen, 1989] T. Kohonen. *Self Organisation and Associative Memory.* Springer Verlag, Berlin, 3rd édition, 1989.

[Korn, 1995] Granino Korn. *Neural Networks and Fuzzy-Logic Control on Personal Computers and Workstations.* MIT Press, 1995.

[Kufrin, 1998] R. Kufrin. An evaluation of barrier synchronization on the Origin2000. Dans *CUG'98 Origin2000 Workshop*, Denver, CO, Octobre 1998.

[Kumar *et al.*, 1994] Vipin Kumar, Shashi Shekhar et Minesh B. Amin. Scalable parallel formulation of the backpropagation algorithm for hypercubes and related architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(10):1073–1090, Octobre 1994.

[Lallement, 1996] Y. Lallement. Intégration neuro-symbolique et intelligence artificielle, applications et implantation parallèle. *PhD. Thesis*, 1996.

[Lamport, 1979] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Septembre 1979.

[Laroche *et al.*, 2000] Pierre Laroche, Yann Boniface et René Schott. Décomposition d'un Processus Décisionnel de Markov à l'aide d'un Graphe. Dans *Rencontres Francophones du Parallélisme (RenPar'2000), Besançon*, Juin 2000.

[le Cun, 1985] Y. le Cun. A learning scheme for asymmetric threshold networks. Dans *Proceedings of Cognitiva 85*, pages 599–604, Paris, France, 1985.

[le Cun, 1987] Y. le Cun. *Modèles Connexionnistes de l'Apprentissage.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 1987.

[LeCun *et al.*, 1990] Yann LeCun, John S. Denker et Sara A. Solla. Optimal brain damage. Dans *nips*, pages 598–605, 1990.

[Lefèvre, 1997] L Lefèvre. *Conception et mise en œuvre d'un environnement de programmation parallèle fondé sur un système de mémoire distribuée virtuellement partagée. Le systeme DOSMOS.* PhD thesis, ENS LYON, 1997.

[Leighton, 1992] R. R. Leighton. *The Aspirin/MIGRAINES Neural Network Software.* The MITRE Corporation, 1992. User's Manual, Release V6.0.

[Li, 1986a] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, Septembre 1986.

[Li, 1986b] Kai Li. Shared virtual memory on loosely coupled multiprocessors. Dans *Proc. IEEE CS 1986, Int. Conf. on Computer Languages*, 1986.

[Linden *et al.*, 1993] A. Linden, Th. Sudbrak et Ch. Tietz. The SESAME Handbook – User Manual. Rapport technique, GMD, Sankt Augustin, 1993.

[Maass et Bishop, 1998] rédacteurs W. Maass et C. Bishop. *Pulsed Neural Networks.* Bradford Book, MIT Press, 1998.

[Margaritis et Evans, 1992] K. G. Margaritis et D. J. Evans. Systolic implementation of neural networks for searching sets of properties. *Parallel Computing*, 18(3):325–334, Mars 1992.

[Martinetz et Schulten, 1991] Thomas Martinetz et Klaus Schulten. A "Neural-Gas" network learns topologies. Dans *Proc. Int. Conf. on Artificial Neural Networks* (Espoo, Finland), rédacteurs T. Kohonen, K. Mäkisara, O. Simula et J. Kangas, volume I, pages 397–402, Amsterdam, Netherlands, 1991. North-Holland.

[Martinetz et Schulten, 1994] Thomas Martinetz et Klaus Schulten. Topology representing networks. *Neural Networks*, 7(2), 1994.

[McCulloch et Pitts, 1943] W.S McCulloch et W.P Pitts. A logical calculus in the ideas immanent in nerveous activity. *Bulletin of Mathematical Biophysics*, 1943.

[Medler, 1998] D. A. Medler. A brief history of connectionism. Dans *Neural Computing Surveys*, volume 1, pages 61–101. 1998. http://www.icsi.berkeley.edu/ jagota/NCS.

[Mentré et Priol, 1998] D. Mentré et T. Priol. Noa: A shared virtual memory over a sci cluster. Dans *Proceedings of SCI Europe'98, Technology and Applications*, pages 43–50. H. Hellwagner, A. Reinefeld, septembre 1998. Bordeaux, France.

[Minski et Papert, 1969] M. Minski et S. Papert. *Perceptron.* M.I.T. Press, Cambridge, Mass., 1969.

[Misra, 1996] M. Misra. Parallel environments for implementing neural network. *Neural Computing Survey*, 1:48–60, 1996.

[Miyata, 1991] Y. Miyata. *A User's Guide to PlaNet.* University of Colorado, Boulder, 1991.

[Mosberger, 1993] D. Mosberger. Memory consistency models. *Operating System Review*, 17, Janvier 1993.

[Mountcastle, 1978] V. B. Mountcastle. An organizing principle for cerebral function. The unit module and the distributed system. Dans *The mindful brain*, Cambridge, 1978. MIT Press.

[NeuralWare, 1989] NeuralWare. *NeuralWorks Professional II Users Guide*, 1989.

[Nichols *et al.*, 1996] B. Nichols, D. Butlar et J. Farrell. *Pthreads Programming.* O'Reilly and Associates, 1996.

[Nordström et Svensson, 1992] T. Nordström et B. Svensson. Using and designing massively parallel computers for artificial neural networks. *Journal of Parallel and Distributed Computing*, 14(3):260–285, 1992.

[Nordström, 1995] Tomas Nordström. *Highly Parallel Computers for Artificial Neural Networks.* PhD thesis, Luleå University of Technology, Luleå, Sweden, 1995.

[Östermark, 1996] Ralf Östermark. A flexible multicomputer algorithm for artificial neural networks. *Neural Networks*, 9(1):169–178, 1996.

[Papadopoulo, 1997] Jean Papadopoulo. Evolution in high-end parallel computing: Smp, clusters, and mpp revisited. Rapport technique, Bull, Juillet 1997. http://www-frec.bull.com/docs/wp_hep.htm.

[Parhami, 1995] Parhami. Simd machines: do they have a significant future ? http://www.ece.ucsb.edu/Faculty/Parhami/FMPC95-SIMD-Panel.html, 1995.

[Paugam-Moisy, 1992] H. Paugam-Moisy. On the convergence of a block-gradient algorithm for back-propagation learning. Dans *IJCNN'92.* International Joint Conference on Neural Networks, 1992.

[Paugam-Moisy, 1995] H. Paugam-Moisy. Multiprocessor simulation of neural networks. Dans *The Handbook of Brain Theory and Neural Network.*, pages 605–608. The MIT Press., 1995.

[Petrowski *et al.*, 1993] Alain Petrowski, Gerard Dreyfus et Claude Girault. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981, Novembre 1993.

[Petrowski, 1993] Alain Petrowski. *Algorithmes parallèles de rétro-propagation des erreurs pour les réseaux de neurones.* PhD thesis, Université Pierre et Marie Curie, mai 1993.

[PGENESIS, 2000] PGENESIS. Pittsburgh Supercomputing Center (PSC), Carnegie Mellon University, 2000. URL: http://www.psc.edu/Packages/PGENESIS/progmodel.html.

[Pican, 1995] N. Pican. *Approche statique et dynamique de la modulation de l'efficacité synaptique dans les réseaux de neurones.* Doctorat d'université, Université Nancy 1, janvier 1995.

[Pican, 1996] N. Pican. Intrinsic and parallel performances of the OWE neural network architecture. *Lecture Notes in Computer Science*, 1112:755–??, 1996.

[Plonski, 90] Plonski. RCS, GENESIS, and SFINX:three public domain simulators for neural networks. *Neural Network Review*, 4(3/4), 90.

[Prechelt, 1994] L. Prechelt. Cupit-a parallel language for neural algorithms : Language reference and tutorial. *Tecnical Report, Univ. Karlsruhe, Allemagne*, 1994.

[Prechelt, 1999] Lutz Prechelt. Exploiting domain-specific properties: Compiling parallel dynamic neural network algorithms into efficient code. *IEEE Transactions on Parallel and Distributed Systems*, 10(11):1105–1117, November 1999.

[Priol, 1997] T. Priol. Mémoire virtuellement partagée pour le calcul de haute performance: évolutions et tendances. *Technique et Science Informatiques*, 16(10):1231–1256, 1997.

[Puzenat, 1997] D. Puzenat. Parallélisme et modularité des modèles connexionnistes. *PhD. Thesis*, 1997.

[Quoy *et al.*, 1997] M. Quoy, O. Gallet et P. Gaussier. Implémentation parallèle d'un système de focalisation de l'attention. Dans *GRETSI*, 1997.

[Reed, 1993] Russel Reed. Pruning algorithms — A survey. *IEEE Transactions on Neural Networks*, 4(5):740–746, 1993.

[Reilly *et al.*, 1987] D. L. Reilly, C. Scofield, C. Elbaum et L. N. Cooper. Learning system architetures composed of multiple learning modules. Dans *Proceedings of the IEEE First International Conference On Neural Networks*, San Diego, 1987.

[Reiss et Taylor, 1991] M. Reiss et J.G Taylor. Strong temporal sequences. *Neural Networks*, 4:773–787, 1991.

[Reski, 1999] Thilo Reski. *Mapping and Parallel, Distributed Simulation of Neural Networks on Message Passing Multiprocessors.* PhD thesis, Universität Paderborn, 1999.

[Rougier, 2000] Nicolas Rougier. *Modèles de mémoires pour la navigation autonome.* PhD thesis, Université Nancy 1, France, 2000.

[Rumelhart *et al.*, 1986] D.E. Rumelhart, J.L. McClelland et the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT Press, Cambridge, 1986.

[SGI, 1998] SGI. *Origin2000 and Onyx2 Performance Tuning and Optimization Guide.* SGI, 10 1998.

[Stenström *et al.*, 1992] P. Stenström, T. Joe et A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. Dans *Proceedings of the 19th Annual International Symposium on Computer Architecture*, rédacteurs David Abramson et Jean-Luc Gaudiot, pages 80–91, Gold Coast, Australia, Mai 1992. ACM Press.

[Stevens, 1978] C. Stevens. *Le neurone.* Le Cerveau, Belin, 1978.

[Strey, 1997] A Strey. EspiloNN - a specification language for the efficient parallel implementation of neural networks. Dans *Biological and Artificial Computation: From Neuroscience to Technology*, rédacteur Cabestany J. Mira J., Moreno-Diaz R., volume 1240 de *Lecture Notes in Computer Science*, pages 714–722, Berlin, 1997. Springer-Verlag.

[Strey, 1999] A Strey. A unified model for the simulation of artificial and biology-oriented neural networks. Dans *Engineering Applications of Bio-Inspired Artificial Neural Networks*, rédacteur Cabestany J. Mira J., Moreno-Diaz R., volume 1607 de *Lecture Notes in Computer Science*, pages 1–10, Berlin, 1999. Springer-Verlag.

[Sundararajan et Saratchandran, 1998] N. Sundararajan et P. Saratchandran. *Parallel architectures for artificial neural network, Paradigms and Implementation.* IEEE computer society, 1998.

[Touretzky, 1990] rédacteur D. S. Touretzky. *Advances in Neural Information Processing Systems 2. Proceedings of the 1989 Conference.* MK, San Mateo, CA, 1990.

[Touzet, 1992] C. Touzet. *Réseaux de neurones artificiels: introduction au connexionnisme (Artificial neural nets: introduction to connectionism).* EC2, Nanterre, France, 1992.

[Vialle *et al.*, 1998] S. Vialle, Y. Lallement et T. Cornu. Design and implementation of a parallel cellular language for mimd architectures. *Computer languages*, 1998.

[Vialle, 1996] S. Vialle. Parcel-1 : un langage parallèle d'acteurs autonomes synchrones. *PhD. Thesis*, 1996.

[von Neumann, 1958] J. von Neumann. *Le cerveau et l'ordinateur.* champ flamarion. Flammarion, 1958.

[Wacquant, 1993] Sylvie Wacquant. *Contribution a l'étude d'un modèle de réseaux d'automates corticaux : Principes et outils logiciels.* PhD thesis, Université de Rouen, 1993.

[Waibel *et al.*, 1989] A. Waibel, T. Hanazawa, G. Hintonand K. Shikano et K. Lang. Phoneme recognition using time delay neural networks. Dans *IEEE Transactions on Acoustics Speech and Signal Processing*, volume 37, 1989.

[Warren et Haridi, 1988] D. H. D. Warren et S. Haridi. Data diffusion machine - A scalable shared virtual memory multiprocessor. Dans *Proceedings of the International Conference on Fifth Generation Computer Systems. Volume 3*, rédacteur Institute for New Generation Computer

Technology (ICOT), pages 943–952, Berlin, FRG, Novembre 28–Décembre 2 1988. Springer Verlag.

[Widrow et Hoff, 1960] B. Widrow et M. E. Hoff. Adaptive switching circuits. Dans *1960 IRE WESCON Convention Record*, volume 4, pages 96–104. IRE, New York, 1960.

[Wilke *et al.*, 1995] P. Wilke, J. Rehder, G. Billing, J. Nilson et C. Mansfield. Neurograph: An integrated development environment for neural networks, genetic algorithms and fuzzy logic. Dans *Third European Congress on Intelligent Techniques and Soft Computing - EUFIT'95*, rédacteur Hans Jürgen Zimmermann, volume 3, pages 1879–1883, Promenade 9, D-52076 Aachen, Août 28-31 1995. Verlag Mainz.

[Williams, 1994] Peter M. Williams. Bayesian regularisation and pruning using a laplace prior. Rapport Technique 312, School of Cognitive and Computing Sciences, University of Sussex, 1994.

[Wilson *et al.*, 1990] Matt Wilson, John Uhley, Upinder Bhalla, David Bilitch, Mark Nelson et James Bower. GENESIS and XODUS, general purpose neural network simulation tool. Dans *Proceedings of the Winter 1990 USENIX Conference, January 22–26, 1990, Washington, DC, USA*, rédacteur USENIX Association, pages 75–88, Berkeley, CA, USA, Janvier 1990. USENIX.

[Zell *et al.*, 1993] A. Zell, H. Bayer, R. Hübner, N. Mache et M. Vogt. Efficient parallel simulation of neural networks. Rapport technique, IPVR University of Stuttgart, Stuttgart, 1993.

[Zell et al, 1993] A. Zell et al. SNNS User Manual, Version 3.0. Rapport technique, Universität Stuttgart, 1993.

[Zhang *et al.*, 1990] Xiru Zhang, M. McKenna, J. P. Mesirov et D. L. Waltz. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, 14(3):317–327, Août 1990.

[Zhang et Yan, 1995] Xiaodong Zhang et Yong Yan. Comparative modeling and evaluation of CC-NUMA and COMA on hierarchical ring architectures. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1316–1331, Décembre 1995.

[Zipser et Andersen, 1988] D. Zipser et R. A. Andersen. A back-propagation programmed network that simulates response properties of a subset of posterior parietal neurons. *Nature*, 331:679–684, 1988.

Annexe A

Un exemple d'implantation : Growing Neural Gas

Nous proposons, dans cette annexe, un exemple d'utilisation de la bibliothèque de simulation. Il s'agit ici de l'implantation de l'algorithme *Growing Neural Gas* [Fritzke, 1995]. Cet algorithme permet la construction dynamique de cartes topologiques.

L'implantation se fait à l'aide de deux types de neurones distincts :

- Un neurone *Maître.*
- Des neurones *Prototypes.*

Les neurones prototypes sont les neurones artificiels de la carte topologique. Ils doivent, au cours de la phase d'apprentissage du réseau, représenter au mieux l'espace des exemples. Ils doivent aussi disposer de liens topologiques reliant les neurones qui représentent des propriétés proches (qui ont un comportement comparable). Chaque neurone *prototype* est uniquement relié au neurone *maître* en terme de connexions neuronales (pour communiquer des informations). Les liens topologiques ne sont que des informations sur le réseau, ils ne fournissent aucune communication entre les différents neurones *prototypes.*

Le neurone maître permet de déterminer, pour chacun des exemples présenté en entrée du réseau, le neurone vainqueur. Il permet de décider de la création de nouveaux neurones et de la position de ces nouveaux neurones dans le réseaux de neurones prototypes. Il est connecté à chacun des neurones *prototypes* de la carte topologique.

L'apprentissage

Le réseau est initialisé avec le neurone *maître* et deux neurones *prototypes.* Le neurone maître est donc relié aux deux neurones *prototypes*, les neurones *prototypes* sont eux-mêmes connectés au neurone maître. De plus, un lien topologique existe entre les deux neurones *prototypes.*

Pour chaque exemple présenté au réseau, le neurone *maître* désigne les deux neurones "vainqueurs", c'est-à-dire les deux neurones *prototypes* pour lesquels les vecteurs de poids sont les plus proches de l'exemple présenté. Le lien topologique entre ces deux neurones est alors créé ou renforcé.

Au cours de l'apprentissage, tous les *lambda* exemples dans notre implantation, le neurone *maître* crée de nouveaux neurones *prototypes.* Chaque nouveau neurone créé est placé entre les

deux neurones ayant accumulé les plus fortes erreurs et se lie, topologiquement, avec ceux-ci, tandis que le lien entre ces deux neurones est supprimé.

Au cours du temps, les neurones remettent à jour leurs liens topologiques en supprimant les liens trop vieux. Un neurone ne disposant plus de lien topologique, étant considéré comme extérieur à la carte, est supprimé (il demande à mourir).

Au terme du traitement d'une certaine taille de carte topologique, l'apprentissage est considéré comme terminé : le neurone *maître* élimine tous les neurones du réseau.

L'implantation

Pour permettre l'implantation de notre réseau, le traitement de chaque exemple est réparti sur trois cycles de vie des neurones. Ces trois cycles sont présentés, pour chacun des deux types de neurones, dans le tableau suivant :

	Pour chaque exemple		
	Cycle 0	*Cycle 1*	*Cycle 1*
neurones prototypes	Calcul distance à l'exemple + Mise a jour (si nouveau neurone)	Mise à jour de l'erreur cumulée (décroissance)	Mise à jour des poids
neurone maître		Détermination des deux neurones vainqueurs	Détermination nouvel exemple + Création neurone

Nous présentons dans la suite le code de l'implantation, en langage C, de l'algorithme. Pour simplifier la lecture, nous ne présentons que les parties de programmes concernant directement la bibliothèque : les fonctions caractéristiques des deux types de neurones et la déclaration du réseau. Il faut noter que les neurones *prototypes* créés au départ du réseau et ceux créés dynamiquement, en cours d'apprentissage, ne diffèrent que par leur fonction d'initialisation.

La déclaration du réseau

```
/* *********************************

   FILE NAME : main_growing.c

***********************************/


#include <GrowingNeuralGas.h>

int main()
{
  int nb_process = 1;
  int i;

  /*
     Initialisation du reseau de
     neurones :
     3 neurones
     'nb_process' processeurs
  */
  InitNetwork(3, nb_process);


  /*
    Creation du neurone 'maitre'
  */
  MakeNeuron(init_master, iter_master, term_master, 0);

  /*
     Creation des deux neurones 'prototypes'
  */
  for(i = 1; i < 3; i ++)
    MakeNeuron(init_firsts, iter_neuron, term_neuron, i);

  /*
     Execution du reseau Growing Neural Gas
  */
  ExecuteNetwork();

  /*
    Menage apres execution
  */
  FreeNetwork();

  return 1;
}
```

L'implantation des neurones *Maître*

```
/**********************************

   FILE NAME : prototype.c

   Fonctions caractéristiques du
   neurone 'maitre' pour
   l'algorithme 'Growing Neural Gas'

**********************************/


#include <neuron.h>

/*
  Type de la sortie
  du maitre
*/
typedef struct {
  int ind;                    /* ==   Indice du prochain exemple a utiliser   == */
  int anc;                    /* ==    Indice du prededent exemple utilise    == */
  int vainqueur;              /* ==   Identite du dernier neurone vainqueur   == */
  int second;                 /* == Identite du dauphin du dernier vainqueur  == */
  char etape;                 /* ==  Permet de connaitre l'etape a effectuer  == */
  int nouv;                   /* ==      Numero du neurone cree si != 0       == */
  float SumE;                 /* ==   Permet d'identifier l'erreur des news   == */
} master_exit;

/*
  Type des variables locales
  du neurone maitre
*/
typedef struct {
  canal_in *lienks;           /* == Tableau de liens, un pour chaque neurones == */
  master_exit exit;           /* ==             Sortie du maitre              == */
  int iter;                   /* ==      Nombre d'iterations effectuees       == */
} type_master;

/*
  Fonction de copie de la
  sortie du neurone maitre
*/
master_exit *CopieMaster(master_exit *mastmod)
{
  master_exit *copiemast;

  copiemast = (master_exit *) malloc (sizeof(master_exit));
  *copiemast = *mastmod;

  return copiemast;
}
```

```
/*
  Fonction d'initialisation
  du neurone maitre
*/
void init_master()
{
  type_master *var;

  /* Allocation de la variable locale            */
  var = (type_master *) malloc(sizeof(type_master));

  /* Declaration de la variable locale
     du neurone au simulateur                    */
  InitLocVar(var);

  /* Initialisation de la variable locale        */
  var->lienks = (canal_in *) malloc(nb_proto_desire * sizeof(canal_in));

  /* Connexion aux deux neurones presents        */
  var->lienks[0] = ConnectAt(1);
  var->lienks[1] = ConnectAt(2);
  var->exit.ind = (int) floor(rand()/(pow(2., 15) -1) * NBPATTERN);
  var->exit.etape = 0;
  var->exit.nouv = 0;
  var->iter = 1;

  /* declaration de la sortie du neurone
     et de la fonction de copie associee         */
  DeclareOutput(&(var->exit), CopieMaster);
}

/*
  Fonction d'iteration
  du neurone maitre
*/
void iter_master()
{
  type_master *var;
  float min;                                  /* == Distance minimum au modele == */
  float sec;                                  /* ==  Seconde distance minimum  == */
  float Eqmax;                                /* ==      Pour la creation      == */
  int indq = -1;
  float Efmax;
  int indf = -1;
  neur_exit *input;
  neur_exit *autre;
  int i;

  /* Recuperation de la variable locale          */
  DecLocVar(var);

  /* Choix de l'activite du neurone en foonction
     de l'etape en cours                         */
  switch (var->exit.etape) {
```

```
case 0:
  if (var→exit.nouv != 0) {
    /* Connexion au nouveau neurone cree        */
    var→lienks[nb_neurons - 1] = ConnectAt(nb_neurons);
    var→exit.nouv = 0;
  }
  break;
case 1:
  /* Recuperation des distances au modeles      */
  /* Determination des 2 neurones vainqueurs    */
  min = FLT_MAX;
  sec = FLT_MAX;

  for (i = 0; i < nb_neurons; i ++) {
    if IsAlive(ReturnOrigin(var→lienks[i])){
      input = InputCanal(neur_exit, var→lienks[i]);

      if (input→sqerror < min) {
        var→exit.second = var→exit.vainqueur;
        var→exit.vainqueur = i +1;
        sec = min;
        min = input→sqerror;
      }

      else
        if (input→sqerror < sec) {
          var→exit.second = i + 1;
          sec = input→sqerror;
        }
    }
  }

  /*          Condition de terminaison         */
  if (nb_neurons == nb_proto_desire - 1) {
    kill_all();
  }
  break;
case 2:
  if (var→iter % lambda == 0) {
    /* Creation d'un nouveau neurone            */

    /* Recherche du neurone a plus forte
       erreur accumulee                         */
    Eqmax = 0;
    for (i = 0; i < nb_neurons; i ++){
      if IsAlive(ReturnOrigin(var→lienks[i])){
        input = InputCanal(neur_exit, var→lienks[i]);
        if (input→Ec > Eqmax) {
          indq = i + 1;
          Eqmax = input→Ec;
        }
      }
    }
```

```
      /* Recherche du neurone voisin a plus
         forte erreur accumulee                    */
      Efmax = 0;
      input = InputCanal(neur_exit, var→lienks[indq -1]);
      for (i = 1; i < nb_neurons + 1; i ++) {
        if IsAlive(ReturnOrigin(var→lienks[i])){
          if ( (input→connects[i]) && (i != indq) ) {
            autre = InputCanal(neur_exit, var→lienks[i-1]);
            if (Efmax < autre→Ec) {
              indf = i;
              Efmax = autre→Ec;
            }
          }
        }
      }
      /* Calculs parametres du nouveau neurone    */
      autre = InputCanal(neur_exit, var→lienks[indf -1]);
      for (i = 0; i < NBPARAM; i ++)
        New_param[i] = .5 * (input→value[i] + autre→value[i]);

      /* creation explicite du nouveau neurone    */
      nb_neurons ++;
      var→exit.nouv = nb_neurons;
      MakeNewNeuron(init_others, iter_neuron, term_neuron, NULL, nb_neurons);

      /* Passage de l'information aux neurones    */
      var→exit.nouv = nb_neurons;
      var→exit.vainqueur = indq;
      var→exit.second = indf;
      var→exit.SumE = 0.5 * (Efmax + Eqmax);
    }
    /* Determination du nouvel exemple           */
    var→exit.ind = DefinedNewExemple();
    /* Incrementation des iterations             */
    var→iter ++;
    break;
  }
  /* Incrementation du compteur d'etapes         */
  var→exit.etape = (++var→exit.etape % 3);
}

/*
  Fonction de terminaison
  du neurone maitre
*/
void term_master()
{
  type_master *var;

  /* Recuperation de la variable locale          */
  DecLocVar(var);
  free(var→lienks);
  free(var);
}
```

L'implantation des neurones *prototypes*

```
/**********************************

   FILE NAME : prototype.c

   Fonctions caractéristiques des
   neurones 'prototypes' pour
   l'algorithme 'Growing Neural Gas'

**********************************/


#include <neuron.h>


/*
  Type de la sortie des neurones
*/
typedef struct {
  float sqerror;                          /* ==      Erreur du neurone       == */
  float Ec;                               /* ==        Erreur cumulee        == */
  float *value;                           /* ==    coordonnees du neurone    == */
  char *connects;                         /* == liens topologiques du neurone == */
} neur_exit;


/*
  Type des variables locales des neurones
*/
typedef struct {
  int id;                                 /* ==      identite du neurone     == */
  canal_in lien;                          /* ==     lien avec le maitre      == */
  char *age;                              /* ==  age d'un lien topologique    == */
  neur_exit exit;                         /* ==      Sortie du neurone       == */
} locales;


/*
  Type de la sortie des neurones
*/
typedef struct {
  int identite;                           /* ==     Identite du neurone      == */
  float distance;                         /* ==    Distance de au modele     == */
} tableau;
```

```
/*
  Fonction de copie de la sortie du neurone
*/
neur_exit *CopieNeuron(neur_exit *modele)
{
  int i;
  neur_exit *copie;

  copie = (neur_exit *) malloc (sizeof(neur_exit));
  copie→sqerror = modele→sqerror;
  copie→Ec = modele→Ec;
  copie→value = (float *) malloc(NBPARAM * sizeof(float));
  for(i = 0; i < NBPARAM; i ++)
    copie→value[i] = modele→value[i];
  copie→connects = (char *) malloc(nb_proto_desire * sizeof(char));
  for(i = 0; i < nb_proto_desire; i ++)
    copie→connects[i] = modele→connects[i];

  return copie;
}

/*
  Fonction d'initialisation des
  neurones crees au depart du reseau
*/
void init_firsts()
{
  locales *var;

  /* Allocation de la variable locale          */
  var = (locales *) malloc (sizeof(locales));
  /* Declaration de la variable locale
     du neurone au simulateur                  */
  InitLocVar(var);

  /* Initialisation de la variable locale      */
  var→exit.value = (float *) malloc(NBPARAM * sizeof(float));
  var→exit.connects = (char *) calloc(nb_proto_desire, sizeof(char));
  var→age = (char *) calloc(nb_proto_desire, sizeof(char));
  var→exit.sqerror = -1;
  var→exit.Ec = 0;
  var→id = me();

  /* Initialisations des poids du neurone      */
  InitWeights(var→exit.value);

  /* Connexion au neurone maitre
     Connexion au neurone 0                    */
  var→lien = ConnectAt(0);

  /* declaration de la sortie du neurone
     et de la fonction de copie associee       */
  DeclareOutput(&(var→exit), CopieNeuron);
}
```

```
/*
  Fonction d'initialisation des neurones
  crees lors de l'execution du reseau
*/
void init_others()
{
  locales *var;
  master_exit *input;
  int i;

  /* Allocation de la variable locale          */
  var = (locales *) malloc (sizeof(locales));

  /* Declaration de la variable locale
     du neurone au simulateur                  */
  InitLocVar(var);

  /* Initialisation de la variable locale      */
  var->exit.value = (float *) malloc(NBPARAM * sizeof(float));
  var->exit.connects = (char *) calloc(nb_proto_desire, sizeof(char));
  var->age = (char *) calloc(nb_proto_desire, sizeof(char));

  var->exit.sqerror = -1;
  var->id = me();

  /* Connexion au neurone maitre
     Connexion au neurone 0                    */
  var->lien = ConnectAt(0);

  /* lecture de l'entree */
  input = InputCanal(master_exit, var->lien);

  var->exit.Ec = input->SumE;

  /* declaration de la sortie du neurone
     et de la fonction de copie associee       */
  DeclareOutput(&(var->exit), CopieNeuron);

  /*   premières connections                   */
  var->exit.connects[input->vainqueur] = 1;
  var->exit.connects[input->second] = 1;
  var->age[input->vainqueur] = 0;
  var->age[input->second] = 0;

  /* Initialisations des poids du neurone      */
  InitWeights( var->exit.value);

  /* Determination de la distance du nouveau
     neurone au modele actuellement traite
     par le reseau                             */
  var->exit.sqerror = 0.0;
  var->exit.sqerror = DetermineDistance(input->ind, var->exit.value);
}
```

```
/*
  Fonction d'iteration des
  neurones prototypes
*/
void iter_neuron()
{
  locales *var;
  master_exit *input;
  int nb_connex;
  int i;

  /* Recuperation de la variable locale          */
  DecLocVar(var);

  /* lecture de l'entree                         */
  input = InputCanal(master_exit, var->lien);

  /* Choix de l'activite du neurone en fonction
     de l'etape en cours                         */
  switch (input->etape) {
  case 0:
    if (input->nouv) {
      /* Un nouveau neurone a ete cree           */

      /* Suppression des liens topologiques entre
         les neurones 'vainqueur' et 'second'    */
      if (var->id == input->vainqueur) {
        /* Suppression du lien topologique
           avec le second                        */
        var->exit.connects[input->second] = 0;

        /* Creation d'un lien topologique  avec
           le nouveau neurone, lien d'age nul    */
        var->exit.connects[input->nouv] = 1;
        var->age[input->nouv] = 0;

        /* Ponderation des erreurs               */
        var->exit.Ec *= (1 - alpha);
      }
      if (var->id == input->second) {
        /* Suppression du lien topologique
           avec le 'vainqueur'                   */
        var->exit.connects[input->vainqueur] = 0;

        /* Creation d'un lien topologique  avec
           le nouveau neurone, lien d'age nul    */
        var->exit.connects[input->nouv] = 1;
        var->age[input->nouv] = 0;

        /* Ponderation des erreurs               */
        var->exit.Ec *= (1 - alpha);
      }
    }
```

```
  /* Calcul de la distance au modele          */
  var→exit.sqerror = 0.0;
  var→exit.sqerror = DetermineDistance(input→ind, var→exit.value);
  break;
case 1:
  /* Decroissance de Ec                       */
  var→exit.Ec *= (1-beta);
  break;
case 2:
  /* Sequence émotion : J'ai gagne ???        */
  if (input→vainqueur == var→id) {
    /* J'AI GAGNE !!! : je renforce mon lien
       topologique avec le second vainqueur    */
    var→exit.connects[input→second] = 1;
    var→age[input→second] = 0;
    var→exit.Ec += var→exit.sqerror;

    /* Mise a jour des poids du neurone        */
    var→exit.value = MiseAJour(input→ind, var→exit.value);

    /*  Mise a jour des liens topologiques     */
      for (i = 1; i < nb_proto_desire; i ++)
        if (var→exit.connects[i])
          var→age[i] ++;
  }
  else if (input→second == var→id) {
    /* Je suis le second                       */
    var→exit.connects[input→vainqueur] = 2;
    var→age[input→vainqueur] = 0;

    /* Mise a jour des poids                   */
    for (i = 0; i < NBPARAM; i ++)
      var→exit.value = MiseAJourSecond(input→ind, var→exit.value);
  }
  /* Tous les neurones verification connexions */
  nb_connex = 0;
  for (i = 1; i < nb_proto_desire; i ++)
    if (var→exit.connects[i] == 1) {
      if (var→age[i] > Tmax) {
        /* Supression connexions trop vieilles */
        var→exit.connects[i] = 0;
        var→age[i] = 0;
      }
      else
        nb_connex ++;
    }
  if (!nb_connex) {
    /* Un neurone sorti de la grille
       topologique est inutile : il meurt.     */
    kill_me();
    var→exit.sqerror = -1;
    var→exit.Ec = 0;
  }
  break;
```

```
  default :
    exit(0);
    break;
  }
}

/*
  Fonction de terminaison
  des neurones prototypes
*/
void term_neuron()
{
  locales *var;

  /* Recuperation de la variable locale          */
  DecLocVar(var);

  /* desallocations de la variable locale        */
  free(var→exit.value);
  free(var→exit.connects);
  free(var→age);
  free(var);
}
```

Monsieur **BONIFACE Yann**

DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 13 NOV. 2000 n° 442

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY 1 — LE PRÉSIDENT

Cl. BURLET

Université Henri Poincaré, Nancy-I
24-30 rue Lionnois - B.P. 3069 - 54013 NANCY CEDEX
Tél. : 03 83 85 48 00 - Fax : 03 83 85 48 48